## NAME

perl − Practical Extraction and Report Language

## SYNOPSIS

**perl** [ **−sTuU** ] [ **−hv** ] [ **−V**[:*configvar*] ]
[ **−cw** ] [ **−d**[:*debugger*] ] [ **−D**[*number/list*] ]
[ **−pna** ] [ **−F***pattern* ] [ **−l**[*octal*] ] [ **−0**[*octal*] ]
[ **−I***dir* ] [ **−m**[**−**]*module* ] [ **−M**[**−**]*'module...'* ] [ **−P** ] [ **−S** ] [ **−x**[*dir*] ]
[ **−i**[*extension*] ] [ **−e** *'command'* ] [ **--** ] [ *programfile* ] [ *argument* ]...

For ease of access, the Perl manual has been split up into a number of sections:

```
perl                Perl overview (this section)
perldelta           Perl changes since previous version
perlfaq             Perl frequently asked questions
perltoc             Perl documentation table of contents

perldata            Perl data structures
perlsyn             Perl syntax
perlop              Perl operators and precedence
perlre              Perl regular expressions
perlrun             Perl execution and options
perlfunc            Perl builtin functions
perlvar             Perl predefined variables
perlsub             Perl subroutines
perlmod             Perl modules: how they work
perlmodlib          Perl modules: how to write and use
perlmodinstall      Perl modules: how to install from CPAN
perlform            Perl formats
perllocale          Perl locale support

perlref             Perl references
perldsc             Perl data structures intro
perllol             Perl data structures: lists of lists
perltoot            Perl OO tutorial
perlobj             Perl objects
perltie             Perl objects hidden behind simple variables
perlbot             Perl OO tricks and examples
perlipc             Perl interprocess communication

perldebug           Perl debugging
perldiag            Perl diagnostic messages
perlsec             Perl security
perltrap            Perl traps for the unwary
perlport            Perl portability guide
perlstyle           Perl style guide

perlpod             Perl plain old documentation
perlbook            Perl book information
```

```
perlembed           Perl ways to embed perl in your C or C++ application
perlapio            Perl internal IO abstraction interface
perlxs              Perl XS application programming interface
perlxstut           Perl XS tutorial
perlguts            Perl internal functions for those doing extensions
perlcall            Perl calling conventions from C

perlhist            Perl history records
```

(If you're intending to read these straight through for the first time, the suggested order will tend to reduce the number of forward references.)

By default, all of the above manpages are installed in the */usr/local/man/* directory.

Extensive additional documentation for Perl modules is available. The default configuration for perl will place this additional documentation in the */usr/local/lib/perl5/man* directory (or else in the *man* subdirectory of the Perl library directory). Some of this additional documentation is distributed standard with Perl, but you'll also find documentation for third-party modules there.

You should be able to view Perl's documentation with your *man*(1) program by including the proper directories in the appropriate start-up files, or in the MANPATH environment variable. To find out where the configuration has installed the manpages, type:

```
perl −V:man.dir
```

If the directories have a common stem, such as */usr/local/man/man1* and */usr/local/man/man3*, you need only to add that stem (*/usr/local/man*) to your *man*(1) configuration files or your MANPATH environment variable. If they do not share a stem, you'll have to add both stems.

If that doesn't work for some reason, you can still use the supplied *perldoc* script to view module information. You might also look into getting a replacement man program.

If something strange has gone wrong with your program and you're not sure where you should look for help, try the **−w** switch first. It will often point out exactly where the trouble is.

## DESCRIPTION

Perl is a language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal).

Perl combines (in the author's opinion, anyway) some of the best features of C, **sed**, **awk**, and **sh**, so people familiar with those languages should have little difficulty with it. (Language historians will also note some vestiges of **csh**, Pascal, and even BASIC−PLUS.) Expression syntax corresponds quite closely to C expression syntax. Unlike most Unix utilities, Perl does not arbitrarily limit the size of your data—if you've got the memory, Perl can slurp in your whole file as a single string. Recursion is of unlimited depth. And the tables used by hashes (previously called ''associative arrays") grow as necessary to prevent degraded performance. Perl uses sophisticated pattern matching techniques to scan large amounts of data very quickly. Although optimized for scanning text, Perl can also deal with binary data, and can make dbm files look like hashes. Setuid Perl scripts are safer than C programs through a dataflow tracing mechanism which prevents many stupid security holes.

If you have a problem that would ordinarily use **sed** or **awk** or **sh**, but it exceeds their capabilities or must run a little faster, and you don't want to write the silly thing in C, then Perl may be for you. There are also translators to turn your **sed** and **awk** scripts into Perl scripts.

But wait, there's more...

Perl version 5 is nearly a complete rewrite, and provides the following additional benefits:

- Many usability enhancements
  It is now possible to write much more readable Perl code (even within regular expressions). Formerly cryptic variable names can be replaced by mnemonic identifiers. Error messages are more informative, and the optional warnings will catch many of the mistakes a novice might make. This cannot be stressed enough. Whenever you get mysterious behavior, try the **–w** switch!!! Whenever you don't get mysterious behavior, try using **–w** anyway.
- Simplified grammar
  The new yacc grammar is one half the size of the old one. Many of the arbitrary grammar rules have been regularized. The number of reserved words has been cut by 2/3. Despite this, nearly all old Perl scripts will continue to work unchanged.
- Lexical scoping
  Perl variables may now be declared within a lexical scope, like "auto" variables in C. Not only is this more efficient, but it contributes to better privacy for "programming in the large". Anonymous subroutines exhibit deep binding of lexical variables (closures).
- Arbitrarily nested data structures
  Any scalar value, including any array element, may now contain a reference to any other variable or subroutine. You can easily create anonymous variables and subroutines. Perl manages your reference counts for you.
- Modularity and reusability
  The Perl library is now defined in terms of modules which can be easily shared among various packages. A package may choose to import all or a portion of a module's published interface. Pragmas (that is, compiler directives) are defined and used by the same mechanism.
- Object-oriented programming
  A package can function as a class. Dynamic multiple inheritance and virtual methods are supported in a straightforward manner and with very little new syntax. Filehandles may now be treated as objects.
- Embeddable and Extensible
  Perl may now be embedded easily in your C or C++ application, and can either call or be called by your routines through a documented interface. The XS preprocessor is provided to make it easy to glue your C or C++ routines into Perl. Dynamic loading of modules is supported, and Perl itself can be made into a dynamic library.
- POSIX compliant
  A major new module is the POSIX module, which provides access to all available POSIX routines and definitions, via object classes where appropriate.
- Package constructors and destructors
  The new BEGIN and END blocks provide means to capture control as a package is being compiled, and after the program exits. As a degenerate case they work just like awk's BEGIN and END when you use the **–p** or **–n** switches.
- Multiple simultaneous DBM implementations
  A Perl program may now access DBM, NDBM, SDBM, GDBM, and Berkeley DB files from the same script simultaneously. In fact, the old dbmopen interface has been generalized to allow any variable to be tied to an object class which defines its access methods.
- Subroutine definitions may now be autoloaded
  In fact, the AUTOLOAD mechanism also allows you to define any arbitrary semantics for undefined subroutine calls. It's not for just autoloading.
- Regular expression enhancements
  You can now specify nongreedy quantifiers. You can now do grouping without creating a backreference. You can now write regular expressions with embedded whitespace and comments for readability. A consistent extensibility mechanism has been added that is upwardly compatible with all old regular expressions.

- Innumerable Unbundled Modules
  The Comprehensive Perl Archive Network described in the *perlmodlib* manpage contains hundreds of plug-and-play modules full of reusable code. See *http://www.perl.com/CPAN* for a site near you.
- Compilability
  While not yet in full production mode, a working perl-to-C compiler does exist. It can generate portable byte code, simple C, or optimized C code.

Okay, that's *definitely* enough hype.

**ENVIRONMENT**

See the *perlrun* manpage.

**AUTHOR**

Larry Wall <*larry@wall.org*>, with the help of oodles of other folks.

If your Perl success stories and testimonials may be of help to others who wish to advocate the use of Perl in their applications, or if you wish to simply express your gratitude to Larry and the Perl developers, please write to <*perl-thanks@perl.org*>.

**FILES**

```
"/tmp/perl-e$$"        temporary file for -e commands
"@INC"                 locations of perl libraries
```

**SEE ALSO**

```
a2p    awk to perl translator

s2p    sed to perl translator
```

**DIAGNOSTICS**

The **–w** switch produces some lovely diagnostics.

See the *perldiag* manpage for explanations of all Perl's diagnostics. The `use diagnostics` pragma automatically turns Perl's normally terse warnings and errors into these longer forms.

Compilation errors will tell you the line number of the error, with an indication of the next token or token type that was to be examined. (In the case of a script passed to Perl via **–e** switches, each **–e** is counted as one line.)

Setuid scripts have additional constraints that can produce error messages such as "Insecure dependency". See the *perlsec* manpage.

Did we mention that you should definitely consider using the **–w** switch?

**BUGS**

The **–w** switch is not mandatory.

Perl is at the mercy of your machine's definitions of various operations such as type casting, *atof()*, and floating-point output with *sprintf()*.

If your stdio requires a seek or eof between reads and writes on a particular stream, so does Perl. (This doesn't apply to *sysread()* and *syswrite()*.)

While none of the built-in data types have any arbitrary size limits (apart from memory size), there are still a few arbitrary limits: a given variable name may not be longer than 255 characters, and no component of your PATH may be longer than 255 if you use **–S**. A regular expression may not compile to more than 32767 bytes internally.

You may mail your bug reports (be sure to include full configuration information as output by the myconfig program in the perl source tree, or by `perl -V`) to <*perlbug@perl.com*>. If you've succeeded in compiling perl, the perlbug script in the utils/ subdirectory can be used to help mail in a bug report.

Perl actually stands for Pathologically Eclectic Rubbish Lister, but don't tell anyone I said that.

**NOTES**

The Perl motto is "There's more than one way to do it." Divining how many more is left as an exercise to the reader.

The three principal virtues of a programmer are Laziness, Impatience, and Hubris. See the Camel Book for why.

**NAME**

perldata − Perl data types

**DESCRIPTION**

**Variable names**

Perl has three data structures: scalars, arrays of scalars, and associative arrays of scalars, known as ‘‘hashes’’. Normal arrays are indexed by number, starting with 0. (Negative subscripts count from the end.) Hash arrays are indexed by string.

Values are usually referred to by name (or through a named reference). The first character of the name tells you to what sort of data structure it refers. The rest of the name tells you the particular value to which it refers. Most often, it consists of a single *identifier*, that is, a string beginning with a letter or underscore, and containing letters, underscores, and digits. In some cases, it may be a chain of identifiers, separated by `::` (or by `'`, but that's deprecated); all but the last are interpreted as names of packages, to locate the namespace in which to look up the final identifier (see the `Packages` entry in the *perlmod* manpage for details). It's possible to substitute for a simple identifier an expression that produces a reference to the value at runtime; this is described in more detail below, and in the *perlref* manpage.

There are also special variables whose names don't follow these rules, so that they don't accidentally collide with one of your normal variables. Strings that match parenthesized parts of a regular expression are saved under names containing only digits after the `$` (see the *perlop* manpage and the *perlre* manpage). In addition, several special variables that provide windows into the inner working of Perl have names containing punctuation characters (see the *perlvar* manpage).

Scalar values are always named with ‘$’, even when referring to a scalar that is part of an array. It works like the English word ‘‘the’’. Thus we have:

```
$days               # the simple scalar value "days"
$days[28]           # the 29th element of array @days
$days{'Feb'}        # the 'Feb' value from hash %days
$#days              # the last index of array @days
```

but entire arrays or array slices are denoted by ‘@’, which works much like the word ‘‘these’’ or ‘‘those":

```
@days               # ($days[0], $days[1],... $days[n])
@days[3,4,5]        # same as @days[3..5]
@days{'a','c'}      # same as ($days{'a'},$days{'c'})
```

and entire hashes are denoted by ‘%’:

```
%days               # (key1, val1, key2, val2 ...)
```

In addition, subroutines are named with an initial ‘&’, though this is optional when it's otherwise unambiguous (just as ‘‘do’’ is often redundant in English). Symbol table entries can be named with an initial ‘*’, but you don't really care about that yet.

Every variable type has its own namespace. You can, without fear of conflict, use the same name for a scalar variable, an array, or a hash (or, for that matter, a filehandle, a subroutine name, or a label). This means that `$foo` and `@foo` are two different variables. It also means that `$foo[1]` is a part of `@foo`, not a part of `$foo`. This may seem a bit weird, but that's okay, because it is weird.

Because variable and array references always start with ‘$’, ‘@’, or ‘%’, the ‘‘reserved’’ words aren't in fact reserved with respect to variable names. (They ARE reserved with respect to labels and filehandles, however, which don't have an initial special character. You can't have a filehandle named ‘‘log’’, for instance. Hint: you could say `open(LOG,'logfile')` rather than `open(log,'logfile')`. Using uppercase filehandles also improves readability and protects you from conflict with future reserved words.) Case *IS* significant— "FOO’’, ‘‘Foo’’, and ‘‘foo’’ are all different names. Names that start with a letter or underscore may also contain digits and underscores.

It is possible to replace such an alphanumeric name with an expression that returns a reference to an object of that type. For a description of this, see the *perlref* manpage.

Names that start with a digit may contain only more digits. Names that do not start with a letter, underscore, or digit are limited to one character, e.g., `$%` or `$$`. (Most of these one character names have a predefined significance to Perl. For instance, `$$` is the current process id.)

### Context

The interpretation of operations and values in Perl sometimes depends on the requirements of the context around the operation or value. There are two major contexts: scalar and list. Certain operations return list values in contexts wanting a list, and scalar values otherwise. (If this is true of an operation it will be mentioned in the documentation for that operation.) In other words, Perl overloads certain operations based on whether the expected return value is singular or plural. (Some words in English work this way, like "fish" and "sheep".)

In a reciprocal fashion, an operation provides either a scalar or a list context to each of its arguments. For example, if you say

```
int( <STDIN> )
```

the integer operation provides a scalar context for the <STDIN> operator, which responds by reading one line from STDIN and passing it back to the integer operation, which will then find the integer value of that line and return that. If, on the other hand, you say

```
sort( <STDIN> )
```

then the sort operation provides a list context for <STDIN>, which will proceed to read every line available up to the end of file, and pass that list of lines back to the sort routine, which will then sort those lines and return them as a list to whatever the context of the sort was.

Assignment is a little bit special in that it uses its left argument to determine the context for the right argument. Assignment to a scalar evaluates the righthand side in a scalar context, while assignment to an array or array slice evaluates the righthand side in a list context. Assignment to a list also evaluates the righthand side in a list context.

User defined subroutines may choose to care whether they are being called in a scalar or list context, but most subroutines do not need to care, because scalars are automatically interpolated into lists. See the `wantarray` entry in the *perlfunc* manpage.

### Scalar values

All data in Perl is a scalar or an array of scalars or a hash of scalars. Scalar variables may contain various kinds of singular data, such as numbers, strings, and references. In general, conversion from one form to another is transparent. (A scalar may not contain multiple values, but may contain a reference to an array or hash containing multiple values.) Because of the automatic conversion of scalars, operations, and functions that return scalars don't need to care (and, in fact, can't care) whether the context is looking for a string or a number.

Scalars aren't necessarily one thing or another. There's no place to declare a scalar variable to be of type "string", or of type "number", or type "filehandle", or anything else. Perl is a contextually polymorphic language whose scalars can be strings, numbers, or references (which includes objects). While strings and numbers are considered pretty much the same thing for nearly all purposes, references are strongly-typed uncastable pointers with builtin reference-counting and destructor invocation.

A scalar value is interpreted as TRUE in the Boolean sense if it is not the null string or the number 0 (or its string equivalent, "0"). The Boolean context is just a special kind of scalar context.

There are actually two varieties of null scalars: defined and undefined. Undefined null scalars are returned when there is no real value for something, such as when there was an error, or at end of file, or when you refer to an uninitialized variable or element of an array. An undefined null scalar may become defined the first time you use it as if it were defined, but prior to that you can use the *defined()* operator to determine whether the value is defined or not.

To find out whether a given string is a valid nonzero number, it's usually enough to test it against both numeric 0 and also lexical "0" (although this will cause **−w** noises). That's because strings that aren't numbers count as 0, just as they do in **awk**:

```
if ($str == 0 && $str ne "0")  {
    warn "That doesn't look like a number";
}
```

That's usually preferable because otherwise you won't treat IEEE notations like `NaN` or `Infinity` properly. At other times you might prefer to use the POSIX::strtod function or a regular expression to check whether data is numeric. See the *perlre* manpage for details on regular expressions.

```
warn "has nondigits"        if     /\D/;
warn "not a natural number" unless /^\d+$/;             # rejects -3
warn "not an integer"       unless /^-?\d+$/;           # rejects +3
warn "not an integer"       unless /^[+-]?\d+$/;
warn "not a decimal number" unless /^-?\d+\.?\d*$/;     # rejects .2
warn "not a decimal number" unless /^-?(?:\d+(?:\.\d*)?|\.\d+)$/;
warn "not a C float"
    unless /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/;
```

The length of an array is a scalar value. You may find the length of array `@days` by evaluating `$#days`, as in **csh**. (Actually, it's not the length of the array, it's the subscript of the last element, because there is (ordinarily) a 0th element.) Assigning to `$#days` changes the length of the array. Shortening an array by this method destroys intervening values. Lengthening an array that was previously shortened *NO LONGER* recovers the values that were in those elements. (It used to in Perl 4, but we had to break this to make sure destructors were called when expected.) You can also gain some miniscule measure of efficiency by pre-extending an array that is going to get big. (You can also extend an array by assigning to an element that is off the end of the array.) You can truncate an array down to nothing by assigning the null list () to it. The following are equivalent:

```
@whatever = ();
$#whatever = -1;
```

If you evaluate a named array in a scalar context, it returns the length of the array. (Note that this is not true of lists, which return the last value, like the C comma operator, nor of built-in functions, which return whatever they feel like returning.) The following is always true:

```
scalar(@whatever) == $#whatever - $[ + 1;
```

Version 5 of Perl changed the semantics of `$[`: files that don't set the value of `$[` no longer need to worry about whether another file changed its value. (In other words, use of `$[` is deprecated.) So in general you can assume that

```
scalar(@whatever) == $#whatever + 1;
```

Some programmers choose to use an explicit conversion so nothing's left to doubt:

```
$element_count = scalar(@whatever);
```

If you evaluate a hash in a scalar context, it returns a value that is true if and only if the hash contains any key/value pairs. (If there are any key/value pairs, the value returned is a string consisting of the number of used buckets and the number of allocated buckets, separated by a slash. This is pretty much useful only to find out whether Perl's (compiled in) hashing algorithm is performing poorly on your data set. For example, you stick 10,000 things in a hash, but evaluating `%HASH` in scalar context reveals "1/16", which means only one out of sixteen buckets has been touched, and presumably contains all 10,000 of your items. This isn't supposed to happen.)

You can preallocate space for a hash by assigning to the *keys()* function. This rounds up the allocated bucked to the next power of two:

```
keys(%users) = 1000;                # allocate 1024 buckets
```

**Scalar value constructors**

Numeric literals are specified in any of the customary floating point or integer formats:

```
12345
12345.67
.23E-10
0xffff              # hex
0377                # octal
4_294_967_296       # underline for legibility
```

String literals are usually delimited by either single or double quotes. They work much like shell quotes: double-quoted string literals are subject to backslash and variable substitution; single-quoted strings are not (except for "`\'`" and "`\\`"). The usual Unix backslash rules apply for making characters such as newline, tab, etc., as well as some more exotic forms. See the section on *Quote and Quotelike Operators* in the *perlop* manpage for a list.

Octal or hex representations in string literals (e.g. '0xffff') are not automatically converted to their integer representation. The *hex()* and *oct()* functions make these conversions for you. See the `hex` entry in the *perlfunc* manpage and the `oct` entry in the *perlfunc* manpage for more details.

You can also embed newlines directly in your strings, i.e., they can end on a different line than they begin. This is nice, but if you forget your trailing quote, the error will not be reported until Perl finds another line containing the quote character, which may be much further on in the script. Variable substitution inside strings is limited to scalar variables, arrays, and array slices. (In other words, names beginning with $ or @, followed by an optional bracketed expression as a subscript.) The following code segment prints out "The price is $100."

```
$Price = '$100';    # not interpreted
print "The price is $Price.\n";     # interpreted
```

As in some shells, you can put curly brackets around the name to delimit it from following alphanumerics. In fact, an identifier within such curlies is forced to be a string, as is any single identifier within a hash subscript. Our earlier example,

```
$days{'Feb'}
```

can be written as

```
$days{Feb}
```

and the quotes will be assumed automatically. But anything more complicated in the subscript will be interpreted as an expression.

Note that a single-quoted string must be separated from a preceding word by a space, because single quote is a valid (though deprecated) character in a variable name (see the `Packages` entry in the *perlmod* manpage).

Three special literals are __FILE__, __LINE__, and __PACKAGE__, which represent the current filename, line number, and package name at that point in your program. They may be used only as separate tokens; they will not be interpolated into strings. If there is no current package (due to an empty `package;` directive), __PACKAGE__ is the undefined value.

The tokens __END__ and __DATA__ may be used to indicate the logical end of the script before the actual end of file. Any following text is ignored, but may be read via a DATA filehandle: main::DATA for

__END__, or PACKNAME::DATA (where PACKNAME is the current package) for __DATA__. The two control characters ^D and ^Z are synonyms for __END__ (or __DATA__ in a module). See the *SelfLoader* manpage for more description of __DATA__, and an example of its use. Note that you cannot read from the DATA filehandle in a BEGIN block: the BEGIN block is executed as soon as it is seen (during compilation), at which point the corresponding __DATA__ (or __END__) token has not yet been seen.

A word that has no other interpretation in the grammar will be treated as if it were a quoted string. These are known as "barewords". As with filehandles and labels, a bareword that consists entirely of lowercase letters risks conflict with future reserved words, and if you use the **−w** switch, Perl will warn you about any such words. Some people may wish to outlaw barewords entirely. If you say

```
use strict 'subs';
```

then any bareword that would NOT be interpreted as a subroutine call produces a compile-time error instead. The restriction lasts to the end of the enclosing block. An inner block may countermand this by saying `no strict 'subs'`.

Array variables are interpolated into double-quoted strings by joining all the elements of the array with the delimiter specified in the `$"` variable (`$LIST_SEPARATOR` in English), space by default. The following are equivalent:

```
$temp = join($",@ARGV);
system "echo $temp";

system "echo @ARGV";
```

Within search patterns (which also undergo double-quotish substitution) there is a bad ambiguity: Is `/$foo[bar]/` to be interpreted as `/${foo}[bar]/` (where `[bar]` is a character class for the regular expression) or as `/${foo[bar]}/` (where `[bar]` is the subscript to array `@foo`)? If `@foo` doesn't otherwise exist, then it's obviously a character class. If `@foo` exists, Perl takes a good guess about `[bar]`, and is almost always right. If it does guess wrong, or if you're just plain paranoid, you can force the correct interpretation with curly brackets as above.

A line-oriented form of quoting is based on the shell "here-doc" syntax. Following a << you specify a string to terminate the quoted material, and all lines following the current line down to the terminating string are the value of the item. The terminating string may be either an identifier (a word), or some quoted text. If quoted, the type of quotes you use determines the treatment of the text, just as in regular quoting. An unquoted identifier works like double quotes. There must be no space between the << and the identifier. (If you put a space it will be treated as a null identifier, which is valid, and matches the first empty line.) The terminating string must appear by itself (unquoted and with no surrounding whitespace) on the terminating line.

```
    print <<EOF;
The price is $Price.
EOF

    print <<"EOF";  # same as above
The price is $Price.
EOF

    print <<`EOC`;  # execute commands
echo hi there
echo lo there
EOC
```

```
    print <<"foo", <<"bar"; # you can stack them
I said foo.
foo
I said bar.
bar

    myfunc(<<"THIS", 23, <<'THAT');
Here's a line
or two.
THIS
and here's another.
THAT
```

Just don't forget that you have to put a semicolon on the end to finish the statement, as Perl doesn't know you're not going to try to do this:

```
    print <<ABC
179231
ABC
    + 20;
```

### List value constructors

List values are denoted by separating individual values by commas (and enclosing the list in parentheses where precedence requires it):

```
(LIST)
```

In a context not requiring a list value, the value of the list literal is the value of the final element, as with the C comma operator. For example,

```
@foo = ('cc', '-E', $bar);
```

assigns the entire list value to array foo, but

```
$foo = ('cc', '-E', $bar);
```

assigns the value of variable bar to variable foo. Note that the value of an actual array in a scalar context is the length of the array; the following assigns the value 3 to `$foo`:

```
@foo = ('cc', '-E', $bar);
$foo = @foo;                # $foo gets 3
```

You may have an optional comma before the closing parenthesis of a list literal, so that you can say:

```
@foo = (
    1,
    2,
    3,
);
```

LISTs do automatic interpolation of sublists. That is, when a LIST is evaluated, each element of the list is evaluated in a list context, and the resulting list value is interpolated into LIST just as if each individual element were a member of LIST. Thus arrays and hashes lose their identity in a LIST—the list

```
(@foo,@bar,&SomeSub,%glarch)
```

contains all the elements of `@foo` followed by all the elements of `@bar`, followed by all the elements returned by the subroutine named SomeSub called in a list context, followed by the key/value pairs of `%glarch`. To make a list reference that does *NOT* interpolate, see the *perlref* manpage.

The null list is represented by (). Interpolating it in a list has no effect. Thus ((),(),()) is equivalent to (). Similarly, interpolating an array with no elements is the same as if no array had been interpolated at that point.

A list value may also be subscripted like a normal array. You must put the list in parentheses to avoid ambiguity. For example:

```
# Stat returns list value.
$time = (stat($file))[8];

# SYNTAX ERROR HERE.
$time = stat($file)[8];  # OOPS, FORGOT PARENTHESES

# Find a hex digit.
$hexdigit = ('a','b','c','d','e','f')[$digit-10];

# A "reverse comma operator".
return (pop(@foo),pop(@foo))[0];
```

You may assign to `undef` in a list. This is useful for throwing away some of the return values of a function:

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

Lists may be assigned to if and only if each element of the list is legal to assign to:

```
($a, $b, $c) = (1, 2, 3);

($map{'red'}, $map{'blue'}, $map{'green'}) = (0x00f, 0x0f0, 0xf00);
```

Array assignment in a scalar context returns the number of elements produced by the expression on the right side of the assignment:

```
$x = (($foo,$bar) = (3,2,1));       # set $x to 3, not 2
$x = (($foo,$bar) = f());           # set $x to f()'s return count
```

This is very handy when you want to do a list assignment in a Boolean context, because most list functions return a null list when finished, which when assigned produces a 0, which is interpreted as FALSE.

The final element may be an array or a hash:

```
($a, $b, @rest) = split;
my($a, $b, %rest) = @_;
```

You can actually put an array or hash anywhere in the list, but the first one in the list will soak up all the values, and anything after it will get a null value. This may be useful in a *local()* or *my()*.

A hash literal contains pairs of values to be interpreted as a key and a value:

```
# same as map assignment above
%map = ('red',0x00f,'blue',0x0f0,'green',0xf00);
```

While literal lists and named arrays are usually interchangeable, that's not the case for hashes. Just because you can subscript a list value like a normal array does not mean that you can subscript a list value as a hash. Likewise, hashes included as parts of other lists (including parameters lists and return lists from functions) always flatten out into key/value pairs. That's why it's good to use references sometimes.

It is often more readable to use the => operator between key/value pairs. The => operator is mostly just a more visually distinctive synonym for a comma, but it also arranges for its left-hand operand to be interpreted as a string— if it's a bareword that would be a legal identifier. This makes it nice for initializing hashes:

```
    %map = (
                 red   => 0x00f,
                 blue  => 0x0f0,
                 green => 0xf00,
   );
```

or for initializing hash references to be used as records:

```
    $rec = {
                witch => 'Mable the Merciless',
                cat   => 'Fluffy the Ferocious',
                date  => '10/31/1776',
    };
```

or for using call-by-named-parameter to complicated functions:

```
   $field = $query->radio_group(
               name      => 'group_name',
               values    => ['eenie','meenie','minie'],
               default   => 'meenie',
               linebreak => 'true',
               labels    => \%labels
   );
```

Note that just because a hash is initialized in that order doesn't mean that it comes out in that order. See the `sort` entry in the *perlfunc* manpage for examples of how to arrange for an output ordering.

## Typeglobs and Filehandles

Perl uses an internal type called a *typeglob* to hold an entire symbol table entry. The type prefix of a typeglob is a `*`, because it represents all types. This used to be the preferred way to pass arrays and hashes by reference into a function, but now that we have real references, this is seldom needed.

The main use of typeglobs in modern Perl is create symbol table aliases. This assignment:

```
    *this = *that;
```

makes `$this` an alias for `$that`, `@this` an alias for `@that`, `%this` an alias for `%that`, &this an alias for &that, etc. Much safer is to use a reference. This:

```
    local *Here::blue = \$There::green;
```

temporarily makes `$Here::blue` an alias for `$There::green`, but doesn't make `@Here::blue` an alias for `@There::green`, or `%Here::blue` an alias for `%There::green`, etc. See the section on *Symbol Tables* in the *perlmod* manpage for more examples of this. Strange though this may seem, this is the basis for the whole module import/export system.

Another use for typeglobs is to to pass filehandles into a function or to create new filehandles. If you need to use a typeglob to save away a filehandle, do it this way:

```
    $fh = *STDOUT;
```

or perhaps as a real reference, like this:

```
    $fh = \*STDOUT;
```

See the *perlsub* manpage for examples of using these as indirect filehandles in functions.

Typeglobs are also a way to create a local filehandle using the *local()* operator. These last until their block is exited, but may be passed back. For example:

```
sub newopen {
    my $path = shift;
    local *FH;  # not my!
    open   (FH, $path)          or  return undef;
    return *FH;
}
$fh = newopen('/etc/passwd');
```

Now that we have the *foo{THING} notation, typeglobs aren't used as much for filehandle manipulations, although they're still needed to pass brand new file and directory handles into or out of functions. That's because *HANDLE{IO} only works if HANDLE has already been used as a handle. In other words, *FH can be used to create new symbol table entries, but *foo{THING} cannot.

Another way to create anonymous filehandles is with the IO::Handle module and its ilk. These modules have the advantage of not hiding different types of the same name during the *local()*. See the bottom of the `open()` entry in the *perlfunc* manpage for an example.

See the *perlref* manpage, the *perlsub* manpage, and the section on *Symbol Tables* in the *perlmod* manpage for more discussion on typeglobs and the *foo{THING} syntax.

**NAME**

perlsyn − Perl syntax

**DESCRIPTION**

A Perl script consists of a sequence of declarations and statements. The only things that need to be declared in Perl are report formats and subroutines. See the sections below for more information on those declarations. All uninitialized user-created objects are assumed to start with a `null` or `0` value until they are defined by some explicit operation such as assignment. (Though you can get warnings about the use of undefined values if you like.) The sequence of statements is executed just once, unlike in **sed** and **awk** scripts, where the sequence of statements is executed for each input line. While this means that you must explicitly loop over the lines of your input file (or files), it also means you have much more control over which files and which lines you look at. (Actually, I'm lying— it is possible to do an implicit loop with either the **−n** or **−p** switch. It's just not the mandatory default like it is in **sed** and **awk**.)

### Declarations

Perl is, for the most part, a free-form language. (The only exception to this is format declarations, for obvious reasons.) Comments are indicated by the `"#"` character, and extend to the end of the line. If you attempt to use `/* */` C−style comments, it will be interpreted either as division or pattern matching, depending on the context, and C++ `//` comments just look like a null regular expression, so don't do that.

A declaration can be put anywhere a statement can, but has no effect on the execution of the primary sequence of statements— declarations all take effect at compile time. Typically all the declarations are put at the beginning or the end of the script. However, if you're using lexically-scoped private variables created with `my()`, you'll have to make sure your format or subroutine definition is within the same block scope as the my if you expect to be able to access those private variables.

Declaring a subroutine allows a subroutine name to be used as if it were a list operator from that point forward in the program. You can declare a subroutine without defining it by saying `sub name`, thus:

```
sub myname;
$me = myname $0             or die "can't get myname";
```

Note that it functions as a list operator, not as a unary operator; so be careful to use `or` instead of `||` in this case. However, if you were to declare the subroutine as `sub myname ($)`, then `myname` would function as a unary operator, so either `or` or `||` would work.

Subroutines declarations can also be loaded up with the `require` statement or both loaded and imported into your namespace with a `use` statement. See the *perlmod* manpage for details on this.

A statement sequence may contain declarations of lexically-scoped variables, but apart from declaring a variable name, the declaration acts like an ordinary statement, and is elaborated within the sequence of statements as if it were an ordinary statement. That means it actually has both compile-time and run-time effects.

### Simple statements

The only kind of simple statement is an expression evaluated for its side effects. Every simple statement must be terminated with a semicolon, unless it is the final statement in a block, in which case the semicolon is optional. (A semicolon is still encouraged there if the block takes up more than one line, because you may eventually add another line.) Note that there are some operators like `eval {}` and `do {}` that look like compound statements, but aren't (they're just TERMs in an expression), and thus need an explicit termination if used as the last item in a statement.

Any simple statement may optionally be followed by a *SINGLE* modifier, just before the terminating semicolon (or block ending). The possible modifiers are:

```
if EXPR
unless EXPR
while EXPR
until EXPR
foreach EXPR
```

The `if` and `unless` modifiers have the expected semantics, presuming you're a speaker of English. The `foreach` modifier is an iterator: For each value in EXPR, it aliases `$_` to the value and executes the statement. The `while` and `until` modifiers have the usual "`while` loop" semantics (conditional evaluated first), except when applied to a `do`−BLOCK (or to the now-deprecated `do`−SUBROUTINE statement), in which case the block executes once before the conditional is evaluated. This is so that you can write loops like:

```
do {
    $line = <STDIN>;
    ...
} until $line  eq ".\n";
```

See the `do` entry in the *perlfunc* manpage. Note also that the loop control statements described later will *NOT* work in this construct, because modifiers don't take loop labels. Sorry. You can always put another block inside of it (for `next`) or around it (for `last`) to do that sort of thing. For `next`, just double the braces:

```
do {{
    next if $x == $y;
    # do something here
}} until $x++ > $z;
```

For `last`, you have to be more elaborate:

```
LOOP: {
        do {
            last if $x = $y**2;
            # do something here
        } while $x++ <= $z;
}
```

### Compound statements

In Perl, a sequence of statements that defines a scope is called a block. Sometimes a block is delimited by the file containing it (in the case of a required file, or the program as a whole), and sometimes a block is delimited by the extent of a string (in the case of an eval).

But generally, a block is delimited by curly brackets, also known as braces. We will call this syntactic construct a BLOCK.

The following compound statements may be used to control flow:

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK
LABEL for (EXPR; EXPR; EXPR) BLOCK
LABEL foreach VAR (LIST) BLOCK
LABEL BLOCK continue BLOCK
```

Note that, unlike C and Pascal, these are defined in terms of BLOCKs, not statements. This means that the curly brackets are *required*--no dangling statements allowed. If you want to write conditionals without curly brackets there are several other ways to do it. The following all do the same thing:

```
if (!open(FOO)) { die "Can't open $FOO: $!"; }
die "Can't open $FOO: $!" unless open(FOO);
open(FOO) or die "Can't open $FOO: $!";     # FOO or bust!
open(FOO) ? 'hi mom' : die "Can't open $FOO: $!";
                    # a bit exotic, that last one
```

The `if` statement is straightforward. Because BLOCKs are always bounded by curly brackets, there is never any ambiguity about which `if` an `else` goes with. If you use `unless` in place of `if`, the sense of the test is reversed.

The `while` statement executes the block as long as the expression is true (does not evaluate to the null string (`""`) or `0` or `"0"`). The LABEL is optional, and if present, consists of an identifier followed by a colon. The LABEL identifies the loop for the loop control statements `next`, `last`, and `redo`. If the LABEL is omitted, the loop control statement refers to the innermost enclosing loop. This may include dynamically looking back your call-stack at run time to find the LABEL. Such desperate behavior triggers a warning if you use the **–w** flag.

If there is a `continue` BLOCK, it is always executed just before the conditional is about to be evaluated again, just like the third part of a `for` loop in C. Thus it can be used to increment a loop variable, even when the loop has been continued via the `next` statement (which is similar to the C `continue` statement).

### Loop Control

The `next` command is like the `continue` statement in C; it starts the next iteration of the loop:

```
LINE: while (<STDIN>) {
    next LINE if /^#/;      # discard comments
    ...
}
```

The `last` command is like the `break` statement in C (as used in loops); it immediately exits the loop in question. The `continue` block, if any, is not executed:

```
LINE: while (<STDIN>) {
    last LINE if /^$/;      # exit when done with header
    ...
}
```

The `redo` command restarts the loop block without evaluating the conditional again. The `continue` block, if any, is *not* executed. This command is normally used by programs that want to lie to themselves about what was just input.

For example, when processing a file like */etc/termcap*. If your input lines might end in backslashes to indicate continuation, you want to skip ahead and get the next record.

```
while (<>) {
    chomp;
    if (s/\\$//) {
        $_ .= <>;
        redo unless eof();
    }
    # now process $_
}
```

which is Perl short-hand for the more explicitly written version:

```
LINE: while (defined($line = <ARGV>)) {
    chomp($line);
    if ($line =~ s/\\$//) {
        $line .= <ARGV>;
        redo LINE unless eof(); # not eof(ARGV)!
    }
    # now process $line
}
```

Note that if there were a `continue` block on the above code, it would get executed even on discarded lines. This is often used to reset line counters or `?pat?` one-time matches.

```
# inspired by :1,$g/fred/s//WILMA/
while (<>) {
    ?(fred)?    && s//WILMA $1 WILMA/;
    ?(barney)?  && s//BETTY $1 BETTY/;
    ?(homer)?   && s//MARGE $1 MARGE/;
} continue {
    print "$ARGV $.: $_";
    close ARGV  if eof();           # reset $.
    reset       if eof();           # reset ?pat?
}
```

If the word `while` is replaced by the word `until`, the sense of the test is reversed, but the conditional is still tested before the first iteration.

The loop control statements don't work in an `if` or `unless`, since they aren't loops. You can double the braces to make them such, though.

```
if (/pattern/) {{
    next if /fred/;
    next if /barney/;
    # so something here
}}
```

The form `while/if BLOCK BLOCK`, available in Perl 4, is no longer available. Replace any occurrence of `if BLOCK` by `if (do BLOCK)`.

### For Loops

Perl's C–style `for` loop works exactly like the corresponding `while` loop; that means that this:

```
for ($i = 1; $i < 10; $i++) {
    ...
}
```

is the same as this:

```
$i = 1;
while ($i < 10) {
    ...
} continue {
    $i++;
}
```

(There is one minor difference: The first form implies a lexical scope for variables declared with `my` in the initialization expression.)

Besides the normal array index looping, `for` can lend itself to many other interesting applications. Here's one that avoids the problem you get into if you explicitly test for end-of-file on an interactive file descriptor causing your program to appear to hang.

```
$on_a_tty = -t STDIN && -t STDOUT;
sub prompt { print "yes? " if $on_a_tty }
for ( prompt(); <STDIN>; prompt() ) {
    # do something
}
```

### Foreach Loops

The `foreach` loop iterates over a normal list value and sets the variable VAR to be each element of the list in turn. If the variable is preceded with the keyword `my`, then it is lexically scoped, and is therefore visible only within the loop. Otherwise, the variable is implicitly local to the loop and regains its former value upon exiting the loop. If the variable was previously declared with `my`, it uses that variable instead of the global one, but it's still localized to the loop. (Note that a lexically scoped variable can cause problems if you have subroutine or format declarations within the loop which refer to it.)

The `foreach` keyword is actually a synonym for the `for` keyword, so you can use `foreach` for readability or `for` for brevity. (Or because the Bourne shell is more familiar to you than *csh*, so writing `for` comes more naturally.) If VAR is omitted, `$_` is set to each value. If any element of LIST is an lvalue, you can modify it by modifying VAR inside the loop. That's because the `foreach` loop index variable is an implicit alias for each item in the list that you're looping over.

If any part of LIST is an array, `foreach` will get very confused if you add or remove elements within the loop body, for example with `splice`. So don't do that.

`foreach` probably won't do what you expect if VAR is a tied or other special variable. Don't do that either.

Examples:

```
for (@ary) { s/foo/bar/ }

foreach my $elem (@elements) {
    $elem *= 2;
}

for $count (10,9,8,7,6,5,4,3,2,1,'BOOM') {
    print $count, "\n"; sleep(1);
}

for (1..15) { print "Merry Christmas\n"; }

foreach $item (split(/:[\\\n:]*/, $ENV{TERMCAP})) {
    print "Item: $item\n";
}
```

Here's how a C programmer might code up a particular algorithm in Perl:

```
for (my $i = 0; $i < @ary1; $i++) {
    for (my $j = 0; $j < @ary2; $j++) {
        if ($ary1[$i] > $ary2[$j]) {
            last; # can't go to outer :-(
        }
        $ary1[$i] += $ary2[$j];
    }
    # this is where that last takes me
}
```

Whereas here's how a Perl programmer more comfortable with the idiom might do it:

```
OUTER: foreach my $wid (@ary1) {
INNER:   foreach my $jet (@ary2) {
            next OUTER if $wid > $jet;
            $wid += $jet;
         }
      }
```

See how much easier this is? It's cleaner, safer, and faster. It's cleaner because it's less noisy. It's safer because if code gets added between the inner and outer loops later on, the new code won't be accidentally executed. The `next` explicitly iterates the other loop rather than merely terminating the inner one. And it's faster because Perl executes a `foreach` statement more rapidly than it would the equivalent `for` loop.

### Basic BLOCKs and Switch Statements

A BLOCK by itself (labeled or not) is semantically equivalent to a loop that executes once. Thus you can use any of the loop control statements in it to leave or restart the block. (Note that this is *NOT* true in `eval{}`, `sub{}`, or contrary to popular belief `do{}` blocks, which do *NOT* count as loops.) The `continue` block is optional.

The BLOCK construct is particularly nice for doing case structures.

```
SWITCH: {
    if (/^abc/) { $abc = 1; last SWITCH; }
    if (/^def/) { $def = 1; last SWITCH; }
    if (/^xyz/) { $xyz = 1; last SWITCH; }
    $nothing = 1;
}
```

There is no official `switch` statement in Perl, because there are already several ways to write the equivalent. In addition to the above, you could write

```
SWITCH: {
    $abc = 1, last SWITCH  if /^abc/;
    $def = 1, last SWITCH  if /^def/;
    $xyz = 1, last SWITCH  if /^xyz/;
    $nothing = 1;
}
```

(That's actually not as strange as it looks once you realize that you can use loop control "operators" within an expression, That's just the normal C comma operator.)

or

```
SWITCH: {
    /^abc/ && do { $abc = 1; last SWITCH; };
    /^def/ && do { $def = 1; last SWITCH; };
    /^xyz/ && do { $xyz = 1; last SWITCH; };
    $nothing = 1;
}
```

or formatted so it stands out more as a “proper” `switch` statement:

```
SWITCH: {
    /^abc/      && do {
                          $abc = 1;
                          last SWITCH;
                     };

    /^def/      && do {
                          $def = 1;
                          last SWITCH;
                     };

    /^xyz/      && do {
                          $xyz = 1;
                          last SWITCH;
                      };
    $nothing = 1;
}
```

or

```
SWITCH: {
    /^abc/ and $abc = 1, last SWITCH;
    /^def/ and $def = 1, last SWITCH;
    /^xyz/ and $xyz = 1, last SWITCH;
    $nothing = 1;
}
```

or even, horrors,

```
if (/^abc/)
    { $abc = 1 }
elsif (/^def/)
    { $def = 1 }
elsif (/^xyz/)
    { $xyz = 1 }
else
    { $nothing = 1 }
```

A common idiom for a `switch` statement is to use `foreach`’s aliasing to make a temporary assignment to `$_` for convenient matching:

```
SWITCH: for ($where) {
            /In Card Names/     && do { push @flags, '-e'; last; };
            /Anywhere/          && do { push @flags, '-h'; last; };
            /In Rulings/        && do {                    last; };
            die "unknown value for form variable where: `$where'";
        }
```

Another interesting approach to a switch statement is arrange for a do block to return the proper value:

```
$amode = do {
    if     ($flag & O_RDONLY) { "r" }       # XXX: isn't this 0?
    elsif  ($flag & O_WRONLY) { ($flag & O_APPEND) ? "a" : "w" }
    elsif  ($flag & O_RDWR)   {
        if ($flag & O_CREAT)  { "w+" }
        else                  { ($flag & O_APPEND) ? "a+" : "r+" }
    }
};
```

Or

```
    print do {
        ($flags & O_WRONLY) ? "write-only"          :
        ($flags & O_RDWR)   ? "read-write"          :
                              "read-only";
    };
```

Or if you are certainly that all the && clauses are true, you can use something like this, which "switches" on the value of the HTTP_USER_AGENT envariable.

```
#!/usr/bin/perl
# pick out jargon file page based on browser
$dir = 'http://www.wins.uva.nl/~mes/jargon';
for ($ENV{HTTP_USER_AGENT}) {
    $page  =    /Mac/            && 'm/Macintrash.html'
             || /Win(dows )?NT/  && 'e/evilandrude.html'
             || /Win|MSIE|WebTV/ && 'm/MicroslothWindows.html'
             || /Linux/          && 'l/Linux.html'
             || /HP-UX/          && 'h/HP-SUX.html'
             || /SunOS/          && 's/ScumOS.html'
             ||                     'a/AppendixB.html';
}
print "Location: $dir/$page\015\012\015\012";
```

That kind of switch statement only works when you know the && clauses will be true. If you don't, the previous ?: example should be used.

You might also consider writing a hash instead of synthesizing a switch statement.

### Goto

Although not for the faint of heart, Perl does support a goto statement. A loop's LABEL is not actually a valid target for a goto; it's just the name of the loop. There are three forms: goto−LABEL, goto−EXPR, and goto−&NAME.

The goto−LABEL form finds the statement labeled with LABEL and resumes execution there. It may not be used to go into any construct that requires initialization, such as a subroutine or a foreach loop. It also can't be used to go into a construct that is optimized away. It can be used to go almost anywhere else within the dynamic scope, including out of subroutines, but it's usually better to use some other construct such as last or die. The author of Perl has never felt the need to use this form of goto (in Perl, that is— C is another matter).

The goto−EXPR form expects a label name, whose scope will be resolved dynamically. This allows for computed gotos per FORTRAN, but isn't necessarily recommended if you're optimizing for maintainability:

```
goto ("FOO", "BAR", "GLARCH")[$i];
```

The `goto-&NAME` form is highly magical, and substitutes a call to the named subroutine for the currently running subroutine. This is used by `AUTOLOAD()` subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place (except that any modifications to `@_` in the current subroutine are propagated to the other subroutine.) After the `goto`, not even `caller()` will be able to tell that this routine was called first.

In almost all cases like this, it's usually a far, far better idea to use the structured control flow mechanisms of `next`, `last`, or `redo` instead of resorting to a `goto`. For certain applications, the catch and throw pair of `eval{}` and *die()* for exception processing can also be a prudent approach.

### PODs: Embedded Documentation

Perl has a mechanism for intermixing documentation with source code. While it's expecting the beginning of a new statement, if the compiler encounters a line that begins with an equal sign and a word, like this

```
=head1 Here There Be Pods!
```

Then that text and all remaining text up through and including a line beginning with `=cut` will be ignored. The format of the intervening text is described in the *perlpod* manpage.

This allows you to intermix your source code and your documentation text freely, as in

```
=item snazzle($)

The snazzle() function will behave in the most spectacular
form that you can possibly imagine, not even excepting
cybernetic pyrotechnics.

=cut back to the compiler, nuff of this pod stuff!

sub snazzle($) {
    my $thingie = shift;
    .........
}
```

Note that pod translators should look at only paragraphs beginning with a pod directive (it makes parsing easier), whereas the compiler actually knows to look for pod escapes even in the middle of a paragraph. This means that the following secret stuff will be ignored by both the compiler and the translators.

```
$a=3;
=secret stuff
 warn "Neither POD nor CODE!?"
=cut back
print "got $a\n";
```

You probably shouldn't rely upon the `warn()` being podded out forever. Not all pod translators are well-behaved in this regard, and perhaps the compiler will become pickier.

One may also use pod directives to quickly comment out a section of code.

### Plain Old Comments (Not!)

Much like the C preprocessor, Perl can process line directives. Using this, one can control Perl's idea of filenames and line numbers in error or warning messages (especially for strings that are processed with `eval()`). The syntax for this mechanism is the same as for most C preprocessors: it matches the regular expression `/^#\s*line\s+(\d+)\s*(?:\s"([^"]*)")?/` with `$1` being the line number for the next line, and `$2` being the optional filename (specified within quotes).

Here are some examples that you should be able to type into your command shell:

```
% perl
# line 200 "bzzzt"
# the `#' on the previous line must be the first char on line
die 'foo';
__END__
foo at bzzzt line 201.

% perl
# line 200 "bzzzt"
eval qq[\n#line 2001 ""\ndie 'foo']; print $@;
__END__
foo at - line 2001.

% perl
eval qq[\n#line 200 "foo bar"\ndie 'foo']; print $@;
__END__
foo at foo bar line 200.

% perl
# line 345 "goop"
eval "\n#line " . __LINE__ . ' "' . __FILE__ ."\"\ndie 'foo'";
print $@;
__END__
foo at goop line 345.
```

**NAME**

perlop − Perl operators and precedence

**SYNOPSIS**

Perl operators have the following associativity and precedence, listed from highest precedence to lowest. Note that all operators borrowed from C keep the same precedence relationship with each other, even where C's precedence is slightly screwy. (This makes learning Perl easier for C folks.) With very few exceptions, these all operate on scalar values only, not array values.

```
left        terms and list operators (leftward)
left        ->
nonassoc    ++ --
right       **
right       ! ~ \ and unary + and -
left        =~ !~
left        * / % x
left        + - .
left        << >>
nonassoc    named unary operators
nonassoc    < > <= >= lt gt le ge
nonassoc    == != <=> eq ne cmp
left        &
left        | ^
left        &&
left        ||
nonassoc    ..  ...
right       ?:
right       = += -= *= etc.
left        , =>
nonassoc    list operators (rightward)
right       not
left        and
left        or xor
```

In the following sections, these operators are covered in precedence order.

Many operators can be overloaded for objects. See the *overload* manpage.

**DESCRIPTION**

## Terms and List Operators (Leftward)

A TERM has the highest precedence in Perl. They includes variables, quote and quote-like operators, any expression in parentheses, and any function whose arguments are parenthesized. Actually, there aren't really functions in this sense, just list operators and unary operators behaving as functions because you put parentheses around the arguments. These are all documented in the *perlfunc* manpage.

If any list operator (*print()*, etc.) or any unary operator (*chdir()*, etc.) is followed by a left parenthesis as the next token, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call.

In the absence of parentheses, the precedence of list operators such as `print`, `sort`, or `chmod` is either very high or very low depending on whether you are looking at the left side or the right side of the operator. For example, in

```
@ary = (1, 3, sort 4, 2);
print @ary;         # prints 1324
```

the commas on the right of the sort are evaluated before the sort, but the commas on the left are evaluated after. In other words, list operators tend to gobble up all the arguments that follow them, and then act like a simple TERM with regard to the preceding expression. Note that you have to be careful with parentheses:

```
# These evaluate exit before doing the print:
print($foo, exit);  # Obviously not what you want.
print $foo, exit;   # Nor is this.

# These do the print before evaluating exit:
(print $foo), exit; # This is what you want.
print($foo), exit;  # Or this.
print ($foo), exit; # Or even this.
```

Also note that

```
print ($foo & 255) + 1, "\n";
```

probably doesn't do what you expect at first glance. See the section on *Named Unary Operators* for more discussion of this.

Also parsed as terms are the `do {}` and `eval {}` constructs, as well as subroutine and method calls, and the anonymous constructors `[]` and `{}`.

See also the section on *Quote and Quote-like Operators* toward the end of this section, as well as the section on *I/O Operators*.

### The Arrow Operator

Just as in C and C++, "`->`" is an infix dereference operator. If the right side is either a `[...]` or `{...}` subscript, then the left side must be either a hard or symbolic reference to an array or hash (or a location capable of holding a hard reference, if it's an lvalue (assignable)). See the *perlref* manpage.

Otherwise, the right side is a method name or a simple scalar variable containing the method name, and the left side must either be an object (a blessed reference) or a class name (that is, a package name). See the *perlobj* manpage.

### Auto-increment and Auto-decrement

"++" and "--" work as in C. That is, if placed before a variable, they increment or decrement the variable before returning the value, and if placed after, increment or decrement the variable after returning the value.

The auto-increment operator has a little extra builtin magic to it. If you increment a variable that is numeric, or that has ever been used in a numeric context, you get a normal increment. If, however, the variable has been used in only string contexts since it was set, and has a value that is not the empty string and matches the pattern `/^[a-zA-Z]*[0-9]*$/`, the increment is done as a string, preserving each character within its range, with carry:

```
print ++($foo = '99');      # prints '100'
print ++($foo = 'a0');      # prints 'a1'
print ++($foo = 'Az');      # prints 'Ba'
print ++($foo = 'zz');      # prints 'aaa'
```

The auto-decrement operator is not magical.

### Exponentiation

Binary "**" is the exponentiation operator. Note that it binds even more tightly than unary minus, so −2**4 is −(2**4), not (−2)**4. (This is implemented using C's *pow*(3) function, which actually works on doubles internally.)

### Symbolic Unary Operators

Unary "!" performs logical negation, i.e., "not". See also `not` for a lower precedence version of this.

Unary "−" performs arithmetic negation if the operand is numeric. If the operand is an identifier, a string consisting of a minus sign concatenated with the identifier is returned. Otherwise, if the string starts with a plus or minus, a string starting with the opposite sign is returned. One effect of these rules is that `-bareword` is equivalent to `"-bareword"`.

Unary "~" performs bitwise negation, i.e., 1's complement. For example, `0666 &~ 027` is 0640. (See also the section on *Integer Arithmetic* and the section on *Bitwise String Operators*.)

Unary "+" has no effect whatsoever, even on strings. It is useful syntactically for separating a function name from a parenthesized expression that would otherwise be interpreted as the complete list of function arguments. (See examples above under the section on *Terms and List Operators (Leftward)*.)

Unary "\" creates a reference to whatever follows it. See the *perlref* manpage. Do not confuse this behavior with the behavior of backslash within a string, although both forms do convey the notion of protecting the next thing from interpretation.

### Binding Operators

Binary "=~" binds a scalar expression to a pattern match. Certain operations search or modify the string `$_` by default. This operator makes that kind of operation work on some other string. The right argument is a search pattern, substitution, or transliteration. The left argument is what is supposed to be searched, substituted, or transliterated instead of the default `$_`. The return value indicates the success of the operation. (If the right argument is an expression rather than a search pattern, substitution, or transliteration, it is interpreted as a search pattern at run time. This can be is less efficient than an explicit search, because the pattern must be compiled every time the expression is evaluated.

Binary "!~" is just like "=~" except the return value is negated in the logical sense.

### Multiplicative Operators

Binary "*" multiplies two numbers.

Binary "/" divides two numbers.

Binary "%" computes the modulus of two numbers. Given integer operands `$a` and `$b`: If `$b` is positive, then `$a % $b` is `$a` minus the largest multiple of `$b` that is not greater than `$a`. If `$b` is negative, then `$a % $b` is `$a` minus the smallest multiple of `$b` that is not less than `$a` (i.e. the result will be less than or equal to zero). Note than when `use integer` is in scope, "%" give you direct access to the modulus operator as implemented by your C compiler. This operator is not as well defined for negative operands, but it will execute faster.

Binary "x" is the repetition operator. In scalar context, it returns a string consisting of the left operand repeated the number of times specified by the right operand. In list context, if the left operand is a list in parentheses, it repeats the list.

```
print '-' x 80;             # print row of dashes

print "\t" x ($tab/8), ' ' x ($tab%8);      # tab over

@ones = (1) x 80;           # a list of 80 1's
@ones = (5) x @ones;        # set all elements to 5
```

### Additive Operators

Binary "+" returns the sum of two numbers.

Binary "−" returns the difference of two numbers.

Binary "." concatenates two strings.

### Shift Operators

Binary "<<" returns the value of its left argument shifted left by the number of bits specified by the right argument. Arguments should be integers. (See also the section on *Integer Arithmetic*.)

Binary ">>" returns the value of its left argument shifted right by the number of bits specified by the right argument. Arguments should be integers. (See also the section on *Integer Arithmetic*.)

### Named Unary Operators

The various named unary operators are treated as functions with one argument, with optional parentheses. These include the filetest operators, like `-f`, `-M`, etc. See the *perlfunc* manpage.

If any list operator (*print()*, etc.) or any unary operator (*chdir()*, etc.) is followed by a left parenthesis as the next token, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call. Examples:

```
chdir $foo    || die;       # (chdir $foo) || die
chdir($foo)   || die;       # (chdir $foo) || die
chdir ($foo)  || die;       # (chdir $foo) || die
chdir +($foo) || die;       # (chdir $foo) || die
```

but, because * is higher precedence than `||`:

```
chdir $foo * 20;    # chdir ($foo * 20)
chdir($foo) * 20;   # (chdir $foo) * 20
chdir ($foo) * 20;  # (chdir $foo) * 20
chdir +($foo) * 20; # chdir ($foo * 20)

rand 10 * 20;       # rand (10 * 20)
rand(10) * 20;      # (rand 10) * 20
rand (10) * 20;     # (rand 10) * 20
rand +(10) * 20;    # rand (10 * 20)
```

See also the section on *Terms and List Operators (Leftward)*.

### Relational Operators

Binary "<" returns true if the left argument is numerically less than the right argument.

Binary ">" returns true if the left argument is numerically greater than the right argument.

Binary "<=" returns true if the left argument is numerically less than or equal to the right argument.

Binary ">=" returns true if the left argument is numerically greater than or equal to the right argument.

Binary "lt" returns true if the left argument is stringwise less than the right argument.

Binary "gt" returns true if the left argument is stringwise greater than the right argument.

Binary "le" returns true if the left argument is stringwise less than or equal to the right argument.

Binary "ge" returns true if the left argument is stringwise greater than or equal to the right argument.

### Equality Operators

Binary "==" returns true if the left argument is numerically equal to the right argument.

Binary "!=" returns true if the left argument is numerically not equal to the right argument.

Binary "<=>" returns −1, 0, or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument.

Binary "eq" returns true if the left argument is stringwise equal to the right argument.

Binary "ne" returns true if the left argument is stringwise not equal to the right argument.

Binary "cmp" returns −1, 0, or 1 depending on whether the left argument is stringwise less than, equal to, or greater than the right argument.

"lt", "le", "ge", "gt" and "cmp" use the collation (sort) order specified by the current locale if `use locale` is in effect. See the *perllocale* manpage.

### Bitwise And

Binary "&" returns its operators ANDed together bit by bit. (See also the section on *Integer Arithmetic* and the section on *Bitwise String Operators*.)

### Bitwise Or and Exclusive Or

Binary "|" returns its operators ORed together bit by bit. (See also the section on *Integer Arithmetic* and the section on *Bitwise String Operators*.)

Binary "^" returns its operators XORed together bit by bit. (See also the section on *Integer Arithmetic* and the section on *Bitwise String Operators*.)

### C−style Logical And

Binary "&&" performs a short-circuit logical AND operation. That is, if the left operand is false, the right operand is not even evaluated. Scalar or list context propagates down to the right operand if it is evaluated.

### C−style Logical Or

Binary "||" performs a short-circuit logical OR operation. That is, if the left operand is true, the right operand is not even evaluated. Scalar or list context propagates down to the right operand if it is evaluated.

The `||` and `&&` operators differ from C's in that, rather than returning 0 or 1, they return the last value evaluated. Thus, a reasonably portable way to find out the home directory (assuming it's not "0") might be:

```
$home = $ENV{'HOME'} || $ENV{'LOGDIR'} ||
    (getpwuid($<))[7] || die "You're homeless!\n";
```

In particular, this means that you shouldn't use this for selecting between two aggregates for assignment:

```
@a = @b || @c;              # this is wrong
@a = scalar(@b) || @c;      # really meant this
@a = @b ? @b : @c;          # this works fine, though
```

As more readable alternatives to `&&` and `||` when used for control flow, Perl provides `and` and `or` operators (see below). The short-circuit behavior is identical. The precedence of "and" and "or" is much lower, however, so that you can safely use them after a list operator without the need for parentheses:

```
unlink "alpha", "beta", "gamma"
        or gripe(), next LINE;
```

With the C−style operators that would have been written like this:

```
unlink("alpha", "beta", "gamma")
        || (gripe(), next LINE);
```

Use "or" for assignment is unlikely to do what you want; see below.

**Range Operators**

Binary ".." is the range operator, which is really two different operators depending on the context. In list context, it returns an array of values counting (by ones) from the left value to the right value. This is useful for writing `foreach (1..10)` loops and for doing slice operations on arrays. In the current implementation, no temporary array is created when the range operator is used as the expression in `foreach` loops, but older versions of Perl might burn a lot of memory when you write something like this:

```
for (1 .. 1_000_000) {
    # code
}
```

In scalar context, ".." returns a boolean value. The operator is bistable, like a flip-flop, and emulates the line-range (comma) operator of **sed**, **awk**, and various editors. Each ".." operator maintains its own boolean state. It is false as long as its left operand is false. Once the left operand is true, the range operator stays true until the right operand is true, *AFTER* which the range operator becomes false again. (It doesn't become false till the next time the range operator is evaluated. It can test the right operand and become false on the same evaluation it became true (as in **awk**), but it still returns true once. If you don't want it to test the right operand till the next evaluation (as in **sed**), use three dots (''...") instead of two.) The right operand is not evaluated while the operator is in the "false" state, and the left operand is not evaluated while the operator is in the "true" state. The precedence is a little lower than || and &&. The value returned is either the empty string for false, or a sequence number (beginning with 1) for true. The sequence number is reset for each range encountered. The final sequence number in a range has the string "E0" appended to it, which doesn't affect its numeric value, but gives you something to search for if you want to exclude the endpoint. You can exclude the beginning point by waiting for the sequence number to be greater than 1. If either operand of scalar ".." is a constant expression, that operand is implicitly compared to the `$.` variable, the current line number. Examples:

As a scalar operator:

```
if (101 .. 200) { print; }  # print 2nd hundred lines
next line if (1 .. /^$/);   # skip header lines
s/^/> / if (/^$/ .. eof()); # quote body

# parse mail messages
while (<>) {
    $in_header =   1  .. /^$/;
    $in_body   = /^$/ .. eof();
    # do something based on those
} continue {
    close ARGV if eof;              # reset $. each file
}
```

As a list operator:

```
for (101 .. 200) { print; } # print $_ 100 times
@foo = @foo[0 .. $#foo];    # an expensive no-op
@foo = @foo[$#foo-4 .. $#foo];      # slice last 5 items
```

The range operator (in list context) makes use of the magical auto-increment algorithm if the operands are strings. You can say

```
@alphabet = ('A' .. 'Z');
```

to get all the letters of the alphabet, or

```
$hexdigit = (0 .. 9, 'a' .. 'f')[$num & 15];
```

to get a hexadecimal digit, or

```
@z2 = ('01' .. '31');  print $z2[$mday];
```

to get dates with leading zeros. If the final value specified is not in the sequence that the magical increment would produce, the sequence goes until the next value would be longer than the final value specified.

**Conditional Operator**

Ternary "?:" is the conditional operator, just as in C. It works much like an if-then-else. If the argument before the ? is true, the argument before the : is returned, otherwise the argument after the : is returned. For example:

```
printf "I have %d dog%s.\n", $n,
        ($n == 1) ? '' : "s";
```

Scalar or list context propagates downward into the 2nd or 3rd argument, whichever is selected.

```
$a = $ok ? $b : $c;  # get a scalar
@a = $ok ? @b : @c;  # get an array
$a = $ok ? @b : @c;  # oops, that's just a count!
```

The operator may be assigned to if both the 2nd and 3rd arguments are legal lvalues (meaning that you can assign to them):

```
($a_or_b ? $a : $b) = $c;
```

This is not necessarily guaranteed to contribute to the readability of your program.

Because this operator produces an assignable result, using assignments without parentheses will get you in trouble. For example, this:

```
$a % 2 ? $a += 10 : $a += 2
```

Really means this:

```
(($a % 2) ? ($a += 10) : $a) += 2
```

Rather than this:

```
($a % 2) ? ($a += 10) : ($a += 2)
```

**Assignment Operators**

"=" is the ordinary assignment operator.

Assignment operators work as in C. That is,

```
$a += 2;
```

is equivalent to

```
$a = $a + 2;
```

although without duplicating any side effects that dereferencing the lvalue might trigger, such as from *tie()*. Other assignment operators work similarly. The following are recognized:

```
**=    +=    *=    &=    <<=    &&=
       -=    /=    |=    >>=    ||=
       .=    %=    ^=
             x=
```

Note that while these are grouped by family, they all have the precedence of assignment.

Unlike in C, the assignment operator produces a valid lvalue. Modifying an assignment is equivalent to doing the assignment and then modifying the variable that was assigned to. This is useful for modifying a copy of something, like this:

```
($tmp = $global) =~ tr [A-Z] [a-z];
```

Likewise,

```
($a += 2) *= 3;
```

is equivalent to

```
$a += 2;
$a *= 3;
```

### Comma Operator

Binary "," is the comma operator. In scalar context it evaluates its left argument, throws that value away, then evaluates its right argument and returns that value. This is just like C's comma operator.

In list context, it's just the list argument separator, and inserts both its arguments into the list.

The => digraph is mostly just a synonym for the comma operator. It's useful for documenting arguments that come in pairs. As of release 5.001, it also forces any word to the left of it to be interpreted as a string.

### List Operators (Rightward)

On the right side of a list operator, it has very low precedence, such that it controls all comma-separated expressions found there. The only operators with lower precedence are the logical operators "and", "or", and "not", which may be used to evaluate calls to list operators without the need for extra parentheses:

```
open HANDLE, "filename"
    or die "Can't open: $!\n";
```

See also discussion of list operators in the section on *Terms and List Operators (Leftward)*.

### Logical Not

Unary "not" returns the logical negation of the expression to its right. It's the equivalent of "!" except for the very low precedence.

### Logical And

Binary "and" returns the logical conjunction of the two surrounding expressions. It's equivalent to && except for the very low precedence. This means that it short-circuits: i.e., the right expression is evaluated only if the left expression is true.

### Logical or and Exclusive Or

Binary "or" returns the logical disjunction of the two surrounding expressions. It's equivalent to || except for the very low precedence. This makes it useful for control flow

```
print FH $data              or die "Can't write to FH: $!";
```

This means that it short-circuits: i.e., the right expression is evaluated only if the left expression is false.

Due to its precedence, you should probably avoid using this for assignment, only for control flow.

```
$a = $b or $c;              # bug: this is wrong
($a = $b) or $c;            # really means this
$a = $b || $c;              # better written this way
```

However, when it's a list context assignment and you're trying to use "||" for control flow, you probably need "or" so that the assignment takes higher precedence.

```
@info = stat($file) || die;     # oops, scalar sense of stat!
@info = stat($file) or die;     # better, now @info gets its due
```

Then again, you could always use parentheses.

Binary "xor" returns the exclusive-OR of the two surrounding expressions. It cannot short circuit, of course.

### C Operators Missing From Perl

Here is what C has that Perl doesn't:

unary & Address-of operator. (But see the "\" operator for taking a reference.)

unary * Dereference-address operator. (Perl's prefix dereferencing operators are typed: $, @, %, and &.)

(TYPE) Type casting operator.

### Quote and Quote-like Operators

While we usually think of quotes as literal values, in Perl they function as operators, providing various kinds of interpolating and pattern matching capabilities. Perl provides customary quote characters for these behaviors, but also provides a way for you to choose your quote character for any of them. In the following table, a {} represents any pair of delimiters you choose. Non-bracketing delimiters use the same character fore and aft, but the 4 sorts of brackets (round, angle, square, curly) will all nest.

```
Customary  Generic        Meaning        Interpolates
    ''       q{}          Literal             no
    ""      qq{}          Literal             yes
    ``      qx{}          Command             yes (unless '' is delimiter)
            qw{}         Word list            no
    //       m{}       Pattern match          yes
            qr{}          Pattern             yes
             s{}{}      Substitution          yes
            tr{}{}    Transliteration         no (but see below)
```

Note that there can be whitespace between the operator and the quoting characters, except when # is being used as the quoting character. `q#foo#` is parsed as being the string `foo`, while `q #foo#` is the operator `q` followed by a comment. Its argument will be taken from the next line. This allows you to write:

```
s {foo}  # Replace foo
  {bar}  # with bar.
```

For constructs that do interpolation, variables beginning with "`$`" or "`@`" are interpolated, as are the following sequences. Within a transliteration, the first ten of these sequences may be used.

```
\t          tab             (HT, TAB)
\n          newline         (NL)
\r          return          (CR)
\f          form feed       (FF)
\b          backspace       (BS)
\a          alarm (bell)    (BEL)
\e          escape          (ESC)
\033        octal char
\x1b        hex char
\c[         control char

\l          lowercase next char
\u          uppercase next char
\L          lowercase till \E
\U          uppercase till \E
\E          end case modification
\Q          quote non-word characters till \E
```

If `use locale` is in effect, the case map used by `\l`, `\L`, `\u` and `\U` is taken from the current locale. See the *perllocale* manpage.

All systems use the virtual `"\n"` to represent a line terminator, called a "newline". There is no such thing as an unvarying, physical newline character. It is an illusion that the operating system, device drivers, C libraries, and Perl all conspire to preserve. Not all systems read `"\r"` as ASCII CR and `"\n"` as ASCII LF. For example, on a Mac, these are reversed, and on systems without line terminator, printing `"\n"` may emit no actual data. In general, use `"\n"` when you mean a "newline" for your system, but use the literal ASCII when you need an exact character. For example, most networking protocols expect and prefer a CR+LF (`"\012\015"` or `"\cJ\cM"`) for line terminators, and although they often accept just `"\012"`, they seldom tolerate just `"\015"`. If you get in the habit of using `"\n"` for networking, you may be burned some day.

You cannot include a literal `$` or `@` within a `\Q` sequence. An unescaped `$` or `@` interpolates the corresponding variable, while escaping will cause the literal string `\$` to be inserted. You'll need to write something like `m/\Quser\E\@\Qhost/`.

Patterns are subject to an additional level of interpretation as a regular expression. This is done as a second pass, after variables are interpolated, so that regular expressions may be incorporated into the pattern from the variables. If this is not what you want, use `\Q` to interpolate a variable literally.

Apart from the above, there are no multiple levels of interpolation. In particular, contrary to the expectations of shell programmers, back-quotes do *NOT* interpolate within double quotes, nor do single quotes impede evaluation of variables when used within double quotes.

### Regexp Quote-Like Operators

Here are the quote-like operators that apply to pattern matching and related activities.

Most of this section is related to use of regular expressions from Perl. Such a use may be considered from two points of view: Perl handles a a string and a "pattern" to RE (regular expression) engine to match, RE engine finds (or does not find) the match, and Perl uses the findings of RE engine for its operation, possibly asking the engine for other matches.

RE engine has no idea what Perl is going to do with what it finds, similarly, the rest of Perl has no idea what a particular regular expression means to RE engine. This creates a clean separation, and in this section we discuss matching from Perl point of view only. The other point of view may be found in the *perlre* manpage.

?PATTERN?

This is just like the `/pattern/` search, except that it matches only once between calls to the *reset()* operator. This is a useful optimization when you want to see only the first occurrence of something in each file of a set of files, for instance. Only `??` patterns local to the current package are reset.

```
while (<>) {
    if (?^$?) {
                        # blank line between header and body
    }
} continue {
    reset if eof;       # clear ?? status for next file
}
```

This usage is vaguely deprecated, and may be removed in some future version of Perl.

m/PATTERN/cgimosx

/PATTERN/cgimosx

Searches a string for a pattern match, and in scalar context returns true (1) or false (''). If no string is specified via the `=~` or `!~` operator, the `$_` string is searched. (The string specified with `=~` need not be an lvalue—it may be the result of an expression evaluation, but remember the `=~` binds rather tightly.) See also the *perlre* manpage. See the *perllocale* manpage for discussion of additional considerations that apply when `use locale` is in effect.

Options are:

```
c   Do not reset search position on a failed match when /g is in effect.
g   Match globally, i.e., find all occurrences.
i   Do case-insensitive pattern matching.
m   Treat string as multiple lines.
o   Compile pattern only once.
s   Treat string as single line.
x   Use extended regular expressions.
```

If "/" is the delimiter then the initial `m` is optional. With the `m` you can use any pair of non-alphanumeric, non-whitespace characters as delimiters (if single quotes are used, no interpretation is done on the replacement string. Unlike Perl 4, Perl 5 treats backticks as normal delimiters; the replacement text is not evaluated as a command). This is particularly useful for matching Unix path names that contain "/", to avoid LTS (leaning toothpick syndrome). If "?" is the delimiter, then the match-only-once rule of `?PATTERN?` applies.

PATTERN may contain variables, which will be interpolated (and the pattern recompiled) every time the pattern search is evaluated. (Note that `$)` and `$|` might not be interpolated because they look like end-of-string tests.) If you want such a pattern to be compiled only once, add a `/o` after the trailing delimiter. This avoids expensive run-time recompilations, and is useful when the value you are interpolating won't change over the life of the script. However, mentioning `/o` constitutes a promise that you won't change the variables in the pattern. If you change them, Perl won't even notice.

If the PATTERN evaluates to the empty string, the last *successfully* matched regular expression is used instead.

If the `/g` option is not used, `m//` in a list context returns a list consisting of the subexpressions matched by the parentheses in the pattern, i.e., (`$1`, `$2`, `$3`...). (Note that here `$1` etc. are also set, and that this differs from Perl 4's behavior.) When there are no parentheses in the pattern, the return value is the list `(1)` for success. With or without parentheses, an empty list is returned upon failure.

Examples:

```
open(TTY, '/dev/tty');
<TTY> =~ /^y/i && foo();    # do foo if desired

if (/Version: *([0-9.]*)/) { $version = $1; }

next if m#^/usr/spool/uucp#;

# poor man's grep
$arg = shift;
while (<>) {
    print if /$arg/o;       # compile only once
}

if (($F1, $F2, $Etc) = ($foo =~ /^(\S+)\s+(\S+)\s*(.*)/))
```

This last example splits `$foo` into the first two words and the remainder of the line, and assigns those three fields to `$F1`, `$F2`, and `$Etc`. The conditional is true if any variables were assigned, i.e., if the pattern matched.

The `/g` modifier specifies global pattern matching—that is, matching as many times as possible within the string. How it behaves depends on the context. In list context, it returns a list of all the substrings matched by all the parentheses in the regular expression. If there are no parentheses, it returns a list of all the matched strings, as if there were parentheses around the whole pattern.

In scalar context, each execution of `m//g` finds the next match, returning TRUE if it matches, and FALSE if there is no further match. The position after the last match can be read or set using the *pos()* function; see the `pos` entry in the *perlfunc* manpage. A failed match normally resets the search position to the beginning of the string, but you can avoid that by adding the `/c` modifier (e.g. `m//gc`). Modifying the target string also resets the search position.

You can intermix `m//g` matches with `m/\G.../g`, where `\G` is a zero-width assertion that matches the exact position where the previous `m//g`, if any, left off. The `\G` assertion is not supported without the `/g` modifier; currently, without `/g`, `\G` behaves just like `\A`, but that's accidental and may change in the future.

Examples:

```
# list context
($one,$five,$fifteen) = (`uptime` =~ /(\d+\.\d+)/g);

# scalar context
$/ = ""; $* = 1;  # $* deprecated in modern perls
while (defined($paragraph = <>)) {
    while ($paragraph =~ /[a-z]['")]*[.!?]+['")]*\s/g) {
        $sentences++;
    }
}
print "$sentences\n";
```

```
# using m//gc with \G
$_ = "ppooqppqq";
while ($i++ < 2) {
    print "1: '";
    print $1 while /(o)/gc; print "', pos=", pos, "\n";
    print "2: '";
    print $1 if /\G(q)/gc;  print "', pos=", pos, "\n";
    print "3: '";
    print $1 while /(p)/gc; print "', pos=", pos, "\n";
}
```

The last example should print:

```
1: 'oo', pos=4
2: 'q', pos=5
3: 'pp', pos=7
1: '', pos=7
2: 'q', pos=8
3: '', pos=8
```

A useful idiom for `lex`–like scanners is `/\G.../gc`. You can combine several regexps like this to process a string part-by-part, doing different actions depending on which regexp matched. Each regexp tries to match where the previous one leaves off.

```
$_ = <<'EOL';
     $url = new URI::URL "http://www/";   die if $url eq "xXx";
EOL
LOOP:
   {
     print(" digits"),         redo LOOP if /\G\d+\b[,.;]?\s*/gc;
     print(" lowercase"),      redo LOOP if /\G[a-z]+\b[,.;]?\s*/gc;
     print(" UPPERCASE"),      redo LOOP if /\G[A-Z]+\b[,.;]?\s*/gc;
     print(" Capitalized"),    redo LOOP if /\G[A-Z][a-z]+\b[,.;]?\s*/gc;
     print(" MiXeD"),          redo LOOP if /\G[A-Za-z]+\b[,.;]?\s*/gc;
     print(" alphanumeric"),   redo LOOP if /\G[A-Za-z0-9]+\b[,.;]?\s*/gc;
     print(" line-noise"),     redo LOOP if /\G[^A-Za-z0-9]+/gc;
     print ". That's all!\n";
   }
```

Here is the output (split into several lines):

```
line-noise lowercase line-noise lowercase UPPERCASE line-noise
UPPERCASE line-noise lowercase line-noise lowercase line-noise
lowercase lowercase line-noise lowercase lowercase line-noise
MiXeD line-noise. That's all!
```

q/STRING/

`'STRING'`

A single-quoted, literal string. A backslash represents a backslash unless followed by the delimiter or another backslash, in which case the delimiter or backslash is interpolated.

```
$foo = q!I said, "You said, 'She said it.'"!;
$bar = q('This is it.');
$baz = '\n';                # a two-character string
```

qq/STRING/

“STRING”

A double-quoted, interpolated string.

```
$_ .= qq
 (*** The previous line contains the naughty word "$1".\n)
            if /(tcl|rexx|python)/;      # :-)
$baz = "\n";                # a one-character string
```

qr/STRING/imosx

A string which is (possibly) interpolated and then compiled as a regular expression. The result may be used as a pattern in a match

```
$re = qr/$pattern/;
$string =~ /foo${re}bar/;   # can be interpolated in other patterns
$string =~ $re;             # or used standalone
```

Options are:

```
i   Do case-insensitive pattern matching.
m   Treat string as multiple lines.
o   Compile pattern only once.
s   Treat string as single line.
x   Use extended regular expressions.
```

The benefit from this is that the pattern is precompiled into an internal representation, and does not need to be recompiled every time a match is attempted. This makes it very efficient to do something like:

```
foreach $pattern (@pattern_list) {
    my $re = qr/$pattern/;
    foreach $line (@lines) {
        if($line =~ /$re/) {
            do_something($line);
        }
    }
}
```

See the *perlre* manpage for additional information on valid syntax for STRING, and for a detailed look at the semantics of regular expressions.

qx/STRING/

‘STRING‘

A string which is (possibly) interpolated and then executed as a system command with `/bin/sh` or its equivalent. Shell wildcards, pipes, and redirections will be honored. The collected standard output of the command is returned; standard error is unaffected. In scalar context, it comes back as a single (potentially multi-line) string. In list context, returns a list of lines (however you've defined lines with $/ or `$INPUT_RECORD_SEPARATOR`).

Because backticks do not affect standard error, use shell file descriptor syntax (assuming the shell supports this) if you care to address this. To capture a command's STDERR and STDOUT together:

```
$output = `cmd 2>&1`;
```

To capture a command's STDOUT but discard its STDERR:

```
$output = `cmd 2>/dev/null`;
```

To capture a command's STDERR but discard its STDOUT (ordering is important here):

```
$output = `cmd 2>&1 1>/dev/null`;
```

To exchange a command's STDOUT and STDERR in order to capture the STDERR but leave its STDOUT to come out the old STDERR:

```
$output = `cmd 3>&1 1>&2 2>&3 3>&-`;
```

To read both a command's STDOUT and its STDERR separately, it's easiest and safest to redirect them separately to files, and then read from those files when the program is done:

```
system("program args 1>/tmp/program.stdout 2>/tmp/program.stderr");
```

Using single-quote as a delimiter protects the command from Perl's double-quote interpolation, passing it on to the shell instead:

```
$perl_info  = qx(ps $$);            # that's Perl's $$
$shell_info = qx'ps $$';            # that's the new shell's $$
```

Note that how the string gets evaluated is entirely subject to the command interpreter on your system. On most platforms, you will have to protect shell metacharacters if you want them treated literally. This is in practice difficult to do, as it's unclear how to escape which characters. See the *perlsec* manpage for a clean and safe example of a manual *fork()* and *exec()* to emulate backticks safely.

On some platforms (notably DOS−like ones), the shell may not be capable of dealing with multiline commands, so putting newlines in the string may not get you what you want. You may be able to evaluate multiple commands in a single line by separating them with the command separator character, if your shell supports that (e.g. `;` on many Unix shells; `&` on the Windows NT `cmd` shell).

Beware that some command shells may place restrictions on the length of the command line. You must ensure your strings don't exceed this limit after any necessary interpolations. See the platform-specific release notes for more details about your particular environment.

Using this operator can lead to programs that are difficult to port, because the shell commands called vary between systems, and may in fact not be present at all. As one example, the `type` command under the POSIX shell is very different from the `type` command under DOS. That doesn't mean you should go out of your way to avoid backticks when they're the right way to get something done. Perl was made to be a glue language, and one of the things it glues together is commands. Just understand what you're getting yourself into.

See the section on *I/O Operators* for more discussion.

qw/STRING/

Returns a list of the words extracted out of STRING, using embedded whitespace as the word delimiters. It is exactly equivalent to

```
split(' ', q/STRING/);
```

This equivalency means that if used in scalar context, you'll get split's (unfortunate) scalar context behavior, complete with mysterious warnings.

Some frequently seen examples:

```
use POSIX qw( setlocale localeconv )
@EXPORT = qw( foo bar baz );
```

A common mistake is to try to separate the words with comma or to put comments into a multi-line `qw`−string. For this reason the `-w` switch produce warnings if the STRING contains the "," or the "#" character.

s/PATTERN/REPLACEMENT/egimosx

Searches a string for a pattern, and if found, replaces that pattern with the replacement text and returns the number of substitutions made. Otherwise it returns false (specifically, the empty string).

If no string is specified via the `=~` or `!~` operator, the `$_` variable is searched and modified. (The string specified with `=~` must be scalar variable, an array element, a hash element, or an assignment to one of those, i.e., an lvalue.)

If the delimiter chosen is single quote, no variable interpolation is done on either the PATTERN or the REPLACEMENT. Otherwise, if the PATTERN contains a $ that looks like a variable rather than an end-of-string test, the variable will be interpolated into the pattern at run-time. If you want the pattern compiled only once the first time the variable is interpolated, use the `/o` option. If the pattern evaluates to the empty string, the last successfully executed regular expression is used instead. See the *perlre* manpage for further explanation on these. See the *perllocale* manpage for discussion of additional considerations that apply when `use locale` is in effect.

Options are:

```
e   Evaluate the right side as an expression.
g   Replace globally, i.e., all occurrences.
i   Do case-insensitive pattern matching.
m   Treat string as multiple lines.
o   Compile pattern only once.
s   Treat string as single line.
x   Use extended regular expressions.
```

Any non-alphanumeric, non-whitespace delimiter may replace the slashes. If single quotes are used, no interpretation is done on the replacement string (the `/e` modifier overrides this, however). Unlike Perl 4, Perl 5 treats backticks as normal delimiters; the replacement text is not evaluated as a command. If the PATTERN is delimited by bracketing quotes, the REPLACEMENT has its own pair of quotes, which may or may not be bracketing quotes, e.g., `s(foo)(bar)` or `s<foo>/bar/`. A `/e` will cause the replacement portion to be interpreted as a full-fledged Perl expression and *eval()*ed right then and there. It is, however, syntax checked at compile-time.

Examples:

```
s/\bgreen\b/mauve/g;                # don't change wintergreen

$path =~ s|/usr/bin|/usr/local/bin|;

s/Login: $foo/Login: $bar/; # run-time pattern

($foo = $bar) =~ s/this/that/;      # copy first, then change

$count = ($paragraph =~ s/Mister\b/Mr./g);  # get change-count

$_ = 'abc123xyz';
s/\d+/$&*2/e;               # yields 'abc246xyz'
s/\d+/sprintf("%5d",$&)/e;  # yields 'abc  246xyz'
s/\w/$& x 2/eg;             # yields 'aabbcc  224466xxyyzz'
```

```
s/%(.)/$percent{$1}/g;      # change percent escapes; no /e
s/%(.)/$percent{$1} || $&/ge;       # expr now, so /e
s/^=(\w+)/&pod($1)/ge;      # use function call

# expand variables in $_, but dynamics only, using
# symbolic dereferencing
s/\$(\w+)/${$1}/g;

# /e's can even nest;  this will expand
# any embedded scalar variable (including lexicals) in $_
s/(\$\w+)/$1/eeg;

# Delete (most) C comments.
$program =~ s {
    /\*     # Match the opening delimiter.
    .*?     # Match a minimal number of characters.
    \*/     # Match the closing delimiter.
} []gsx;

s/^\s*(.*?)\s*$/$1/;        # trim white space in $_, expensively

for ($variable) {           # trim white space in $variable, cheap
    s/^\s+//;
    s/\s+$//;
}

s/([^ ]*) *([^ ]*)/$2 $1/;  # reverse 1st two fields
```

Note the use of $ instead of \ in the last example. Unlike **sed**, we use the \<*digit*> form in only the left hand side. Anywhere else it's $<*digit*>.

Occasionally, you can't use just a `/g` to get all the changes to occur. Here are two common cases:

```
# put commas in the right places in an integer
1 while s/(.*\d)(\d\d\d)/$1,$2/g;      # perl4
1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/g;  # perl5

# expand tabs to 8-column spacing
1 while s/\t+/' ' x (length($&)*8 - length($`)%8)/e;
```

tr/SEARCHLIST/REPLACEMENTLIST/cds

y/SEARCHLIST/REPLACEMENTLIST/cds

Transliterates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced or deleted. If no string is specified via the =~ or !~ operator, the `$_` string is transliterated. (The string specified with =~ must be a scalar variable, an array element, a hash element, or an assignment to one of those, i.e., an lvalue.) A character range may be specified with a hyphen, so `tr/A-J/0-9/` does the same replacement as `tr/ACEGIBDFHJ/0246813579/`. For **sed** devotees, `y` is provided as a synonym for `tr`. If the SEARCHLIST is delimited by bracketing quotes, the REPLACEMENTLIST has its own pair of quotes, which may or may not be bracketing quotes, e.g., `tr[A-Z][a-z]` or `tr(+\-*/)/ABCD/`.

Options:

```
c   Complement the SEARCHLIST.
d   Delete found but unreplaced characters.
s   Squash duplicate replaced characters.
```

If the `/c` modifier is specified, the SEARCHLIST character set is complemented. If the `/d` modifier is specified, any characters specified by SEARCHLIST not found in REPLACEMENTLIST are deleted. (Note that this is slightly more flexible than the behavior of some **tr** programs, which delete anything they find in the SEARCHLIST, period.) If the `/s` modifier is specified, sequences of characters that were transliterated to the same character are squashed down to a single instance of the character.

If the `/d` modifier is used, the REPLACEMENTLIST is always interpreted exactly as specified. Otherwise, if the REPLACEMENTLIST is shorter than the SEARCHLIST, the final character is replicated till it is long enough. If the REPLACEMENTLIST is empty, the SEARCHLIST is replicated. This latter is useful for counting characters in a class or for squashing character sequences in a class.

Examples:

```
$ARGV[1] =~ tr/A-Z/a-z/;    # canonicalize to lower case

$cnt = tr/*/*/;             # count the stars in $_

$cnt = $sky =~ tr/*/*/;     # count the stars in $sky

$cnt = tr/0-9//;            # count the digits in $_

tr/a-zA-Z//s;               # bookkeeper -> bokeper

($HOST = $host) =~ tr/a-z/A-Z/;

tr/a-zA-Z/ /cs;             # change non-alphas to single space

tr [\200-\377]
   [\000-\177];             # delete 8th bit
```

If multiple transliterations are given for a character, only the first one is used:

```
tr/AAA/XYZ/
```

will transliterate any A to X.

Note that because the transliteration table is built at compile time, neither the SEARCHLIST nor the REPLACEMENTLIST are subjected to double quote interpolation. That means that if you want to use variables, you must use an *eval()*:

```
eval "tr/$oldlist/$newlist/";
die $@ if $@;

eval "tr/$oldlist/$newlist/, 1" or die $@;
```

### Gory details of parsing quoted constructs

When presented with something which may have several different interpretations, Perl uses the principle **DWIM** (expanded to Do What I Mean – not what I wrote) to pick up the most probable interpretation of the source. This strategy is so successful that Perl users usually do not suspect ambivalence of what they write. However, time to time Perl's ideas differ from what the author meant.

The target of this section is to clarify the Perl's way of interpreting quoted constructs. The most frequent reason one may have to want to know the details discussed in this section is hairy regular expressions. However, the first steps of parsing are the same for all Perl quoting operators, so here they are discussed together.

Some of the passes discussed below are performed concurrently, but as far as results are the same, we consider them one-by-one. For different quoting constructs Perl performs different number of passes, from one to five, but they are always performed in the same order.

Finding the end

First pass is finding the end of the quoted construct, be it multichar ender `"\nEOF\n"` of `<<EOF` construct, `/` which terminates `qq/` construct, `]` which terminates `qq[` construct, or `>` which terminates a fileglob started with `<`.

When searching for multichar construct no skipping is performed. When searching for one-char non-matching delimiter, such as `/`, combinations `\\` and `\/` are skipped. When searching for one-char matching delimiter, such as `]`, combinations `\\`, `\]` and `\[` are skipped, and nested `[`, `]` are skipped as well.

For 3-parts constructs, `s///` etc. the search is repeated once more.

During this search no attention is paid to the semantic of the construct, thus

```
"$hash{"$foo/$bar"}"
```

or

```
m/
  bar       #  This is not a comment, this slash / terminated m//!
 /x
```

do not form legal quoted expressions. Note that since the slash which terminated `m//` was followed by a `SPACE`, this is not `m//x`, thus `#` was interpreted as a literal `#`.

Removal of backslashes before delimiters

During the second pass the text between the starting delimiter and the ending delimiter is copied to a safe location, and the `\` is removed from combinations consisting of `\` and *delimiter*(s) (both starting and ending delimiter if they differ).

The removal does not happen for multi-char delimiters.

Note that the combination `\\` is left as it was!

Starting from this step no information about the *delimiter*(s) is used in the parsing.

Interpolation

Next step is interpolation in the obtained delimiter-independent text. There are four different cases.

`<<'EOF'`, `m''`, `s'''`, `tr///`, `y///`

No interpolation is performed.

`''`, `q//`
The only interpolation is removal of `\` from pairs `\\`.

`""`, ` `` `, `qq//`, `qx//`, `<file*glob>`

`\Q`, `\U`, `\u`, `\L`, `\l` (possibly paired with `\E`) are converted to corresponding Perl constructs, thus `"$foo\Qbaz$bar"` is converted to

```
$foo . (quotemeta("baz" . $bar));
```

Other combinations of `\` with following chars are substituted with appropriate expansions.

Interpolated scalars and arrays are converted to `join` and `.` Perl constructs, thus `"'@arr'"` becomes

```
"'" . (join $", @arr) . "'";
```

Since all three above steps are performed simultaneously left-to-right, the is no way to insert a literal `$` or `@` inside `\Q\E` pair: it cannot be protected by `\`, since any `\` (except in `\E`) is interpreted as a literal inside `\Q\E`, and any `$` is interpreted as starting an interpolated scalar.

Note also that the interpolating code needs to make decision where the interpolated scalar ends, say, whether `"a $b -> {c}"` means

```
"a " . $b . " -> {c}";
```

or

```
"a " . $b -> {c};
```

Most the time the decision is to take the longest possible text which does not include spaces between components and contains matching braces/brackets.

`?RE?`, `/RE/`, `m/RE/`, `s/RE/foo/`,
Processing of `\Q`, `\U`, `\u`, `\L`, `\l` and interpolation happens (almost) as with `qq//` constructs, but *the substitution of* `\` followed by other chars is not performed! Moreover, inside `(?{BLOCK})` no processing is performed at all.

Interpolation has several quirks: `$|`, `$(` and `$)` are not interpolated, and constructs `$var[SOMETHING]` are *voted* (by several different estimators) to be an array element or `$var` followed by a RE alternative. This is the place where the notation `${arr[$bar]}` comes handy: `/${arr[0-9]}/` is interpreted as an array element `-9`, not as a regular expression from variable `$arr` followed by a digit, which is the interpretation of `/$arr[0-9]/`.

Note that absence of processing of `\\` creates specific restrictions on the post-processed text: if the delimiter is `/`, one cannot get the combination `\/` into the result of this step: `/` will finish the regular expression, `\/` will be stripped to `/` on the previous step, and `\\/` will be left as is. Since `/` is equivalent to `\/` inside a regular expression, this does not matter unless the delimiter is special character for the RE engine, as in `s*foo*bar*`, `m[foo]`, or `?foo?`.

This step is the last one for all the constructs except regular expressions, which are processed further.

Interpolation of regular expressions
All the previous steps were performed during the compilation of Perl code, this one happens in run time (though it may be optimized to be calculated at compile time if appropriate). After all the preprocessing performed above (and possibly after evaluation if catenation, joining, up/down-casing and `quotemeta()`ing are involved) the resulting *string* is passed to RE engine for compilation.

Whatever happens in the RE engine is better be discussed in the *perlre* manpage, but for the sake of continuity let us do it here.

This is the first step where presence of the `//x` switch is relevant. The RE engine scans the string left-to-right, and converts it to a finite automaton.

Backslashed chars are either substituted by corresponding literal strings, or generate special nodes of the finite automaton. Characters which are special to the RE engine generate corresponding nodes. `(?#...)` comments are ignored. All the rest is either converted to literal strings to match, or is ignored (as is whitespace and `#`−style comments if `//x` is present).

Note that the parsing of the construct `[...]` is performed using absolutely different rules than the rest of the regular expression. Similarly, the `(?{...})` is only checked for matching braces.

Optimization of regular expressions
: This step is listed for completeness only. Since it does not change semantics, details of this step are not documented and are subject to change.

### I/O Operators

There are several I/O operators you should know about. A string enclosed by backticks (grave accents) first undergoes variable substitution just like a double quoted string. It is then interpreted as a command, and the output of that command is the value of the pseudo-literal, like in a shell. In scalar context, a single string consisting of all the output is returned. In list context, a list of values is returned, one for each line of output. (You can set `$/` to use a different line terminator.) The command is executed each time the pseudo-literal is evaluated. The status value of the command is returned in `$?` (see the *perlvar* manpage for the interpretation of `$?`). Unlike in **csh**, no translation is done on the return data— newlines remain newlines. Unlike in any of the shells, single quotes do not hide variable names in the command from interpretation. To pass a $ through to the shell you need to hide it with a backslash. The generalized form of backticks is `qx//`. (Because backticks always undergo shell expansion as well, see the *perlsec* manpage for security concerns.)

Evaluating a filehandle in angle brackets yields the next line from that file (newline, if any, included), or `undef` at end of file. Ordinarily you must assign that value to a variable, but there is one situation where an automatic assignment happens. *If and ONLY if* the input symbol is the only thing inside the conditional of a `while` or `for(;;)` loop, the value is automatically assigned to the variable `$_`. In these loop constructs, the assigned value (whether assignment is automatic or explicit) is then tested to see if it is defined. The defined test avoids problems where line has a string value that would be treated as false by perl e.g. "" or "0" with no trailing newline. (This may seem like an odd thing to you, but you'll use the construct in almost every Perl script you write.) Anyway, the following lines are equivalent to each other:

```
while (defined($_ = <STDIN>)) { print; }
while ($_ = <STDIN>) { print; }
while (<STDIN>) { print; }
for (;<STDIN>;) { print; }
print while defined($_ = <STDIN>);
print while ($_ = <STDIN>);
print while <STDIN>;
```

and this also behaves similarly, but avoids the use of `$_` :

```
while (my $line = <STDIN>) { print $line }
```

If you really mean such values to terminate the loop they should be tested for explicitly:

```
while (($_ = <STDIN>) ne '0') { ... }
while (<STDIN>) { last unless $_; ... }
```

In other boolean contexts, <*filehandle*> without explicit `defined` test or comparison will solicit a warning if `-w` is in effect.

The filehandles STDIN, STDOUT, and STDERR are predefined. (The filehandles `stdin`, `stdout`, and `stderr` will also work except in packages, where they would be interpreted as local identifiers rather than global.) Additional filehandles may be created with the *open()* function. See the `open()` entry in the *perlfunc* manpage for details on this.

If a <FILEHANDLE> is used in a context that is looking for a list, a list consisting of all the input lines is returned, one line per list element. It's easy to make a *LARGE* data space this way, so use with care.

The null filehandle <> is special and can be used to emulate the behavior of **sed** and **awk**. Input from <> comes either from standard input, or from each file listed on the command line. Here's how it works: the first time <> is evaluated, the `@ARGV` array is checked, and if it is empty, `$ARGV[0]` is set to "−", which when opened gives you standard input. The `@ARGV` array is then processed as a list of filenames. The loop

```
while (<>) {
    ...                         # code for each line
}
```

is equivalent to the following Perl-like pseudo code:

```
unshift(@ARGV, '-') unless @ARGV;
while ($ARGV = shift) {
    open(ARGV, $ARGV);
    while (<ARGV>) {
        ...         # code for each line
    }
}
```

except that it isn't so cumbersome to say, and will actually work. It really does shift array `@ARGV` and put the current filename into variable `$ARGV`. It also uses filehandle *ARGV* internally--<> is just a synonym for <ARGV>, which is magical. (The pseudo code above doesn't work because it treats <ARGV> as non-magical.)

You can modify `@ARGV` before the first <> as long as the array ends up containing the list of filenames you really want. Line numbers (`$.`) continue as if the input were one big happy file. (But see example under `eof` for how to reset line numbers on each file.)

If you want to set `@ARGV` to your own list of files, go right ahead. This sets `@ARGV` to all plain text files if no `@ARGV` was given:

```
@ARGV = grep { -f && -T } glob('*') unless @ARGV;
```

You can even set them to pipe commands. For example, this automatically filters compressed arguments through **gzip**:

```
@ARGV = map { /\.(gz|Z)$/ ? "gzip -dc < $_ |" : $_ } @ARGV;
```

If you want to pass switches into your script, you can use one of the Getopts modules or put a loop on the front like this:

```
while ($_ = $ARGV[0], /^-/) {
    shift;
    last if /^--$/;
    if (/^-D(.*)/) { $debug = $1 }
    if (/^-v/)     { $verbose++  }
    # ...           # other switches
}

while (<>) {
    # ...           # code for each line
}
```

The <> symbol will return `undef` for end-of-file only once. If you call it again after this it will assume you are processing another `@ARGV` list, and if you haven't set `@ARGV`, will input from STDIN.

If the string inside the angle brackets is a reference to a scalar variable (e.g., <$foo>), then that variable contains the name of the filehandle to input from, or its typeglob, or a reference to the same. For example:

```
$fh = \*STDIN;
$line = <$fh>;
```

If what's within the angle brackets is neither a filehandle nor a simple scalar variable containing a filehandle name, typeglob, or typeglob reference, it is interpreted as a filename pattern to be globbed, and either a list of filenames or the next filename in the list is returned, depending on context. This distinction is determined on syntactic grounds alone. That means `<$x>` is always a readline from an indirect handle, but `<$hash{key}>` is always a glob. That's because `$x` is a simple scalar variable, but `$hash{key}` is not— it's a hash element.

One level of double-quote interpretation is done first, but you can't say `<$foo>` because that's an indirect filehandle as explained in the previous paragraph. (In older versions of Perl, programmers would insert curly brackets to force interpretation as a filename glob: `<${foo}>`. These days, it's considered cleaner to call the internal function directly as `glob($foo)`, which is probably the right way to have done it in the first place.) Example:

```
while (<*.c>) {
    chmod 0644, $_;
}
```

is equivalent to

```
open(FOO, "echo *.c | tr -s ' \t\r\f' '\\012\\012\\012\\012'|");
while (<FOO>) {
    chop;
    chmod 0644, $_;
}
```

In fact, it's currently implemented that way. (Which means it will not work on filenames with spaces in them unless you have *csh*(1) on your machine.) Of course, the shortest way to do the above is:

```
chmod 0644, <*.c>;
```

Because globbing invokes a shell, it's often faster to call *readdir()* yourself and do your own *grep()* on the filenames. Furthermore, due to its current implementation of using a shell, the *glob()* routine may get "Arg list too long" errors (unless you've installed *tcsh*(1L) as */bin/csh*).

A glob evaluates its (embedded) argument only when it is starting a new list. All values must be read before it will start over. In a list context this isn't important, because you automatically get them all anyway. In scalar context, however, the operator returns the next value each time it is called, or a `undef` value if you've just run out. As for filehandles an automatic `defined` is generated when the glob occurs in the test part of a `while` or `for` – because legal glob returns (e.g. a file called *0*) would otherwise terminate the loop. Again, `undef` is returned only once. So if you're expecting a single value from a glob, it is much better to say

```
($file) = <blurch*>;
```

than

```
$file = <blurch*>;
```

because the latter will alternate between returning a filename and returning FALSE.

It you're trying to do variable interpolation, it's definitely better to use the *glob()* function, because the older notation can cause people to become confused with the indirect filehandle notation.

```
@files = glob("$dir/*.[ch]");
@files = glob($files[$i]);
```

### Constant Folding

Like C, Perl does a certain amount of expression evaluation at compile time, whenever it determines that all arguments to an operator are static and have no side effects. In particular, string concatenation happens at compile time between literals that don't do variable substitution. Backslash interpretation also happens at compile time. You can say

```
'Now is the time for all' . "\n" .
    'good men to come to.'
```

and this all reduces to one string internally. Likewise, if you say

```
foreach $file (@filenames) {
    if (-s $file > 5 + 100 * 2**16) {  }
}
```

the compiler will precompute the number that expression represents so that the interpreter won't have to.

### Bitwise String Operators

Bitstrings of any size may be manipulated by the bitwise operators (`~ | & ^`).

If the operands to a binary bitwise op are strings of different sizes, **or** and **xor** ops will act as if the shorter operand had additional zero bits on the right, while the **and** op will act as if the longer operand were truncated to the length of the shorter.

```
# ASCII-based examples
print "j p \n" ^ " a h";            # prints "JAPH\n"
print "JA" | "  ph\n";              # prints "japh\n"
print "japh\nJunk" & '_____';       # prints "JAPH\n";
print 'p N$' ^ " E<H\n";            # prints "Perl\n";
```

If you are intending to manipulate bitstrings, you should be certain that you're supplying bitstrings: If an operand is a number, that will imply a **numeric** bitwise operation. You may explicitly show which type of operation you intend by using `""` or `0+`, as in the examples below.

```
$foo =  150  |  105 ;       # yields 255  (0x96 | 0x69 is 0xFF)
$foo = '150' |  105 ;       # yields 255
$foo =  150  | '105';       # yields 255
$foo = '150' | '105';       # yields string '155' (under ASCII)

$baz = 0+$foo & 0+$bar;     # both ops explicitly numeric
$biz = "$foo" ^ "$bar";     # both ops explicitly stringy
```

### Integer Arithmetic

By default Perl assumes that it must do most of its arithmetic in floating point. But by saying

```
use integer;
```

you may tell the compiler that it's okay to use integer operations from here to the end of the enclosing BLOCK. An inner BLOCK may countermand this by saying

```
no integer;
```

which lasts until the end of that BLOCK.

The bitwise operators ("&", "|", "^", "~", "<<", and ">>") always produce integral results. (But see also the section on *Bitwise String Operators*.) However, `use integer` still has meaning for them. By default, their results are interpreted as unsigned integers. However, if `use integer` is in effect, their results are interpreted as signed integers. For example, `~0` usually evaluates to a large integral value. However, `use integer; ~0` is −1 on twos-complement machines.

**Floating-point Arithmetic**

While `use integer` provides integer-only arithmetic, there is no similar ways to provide rounding or truncation at a certain number of decimal places. For rounding to a certain number of digits, *sprintf()* or *printf()* is usually the easiest route.

Floating-point numbers are only approximations to what a mathematician would call real numbers. There are infinitely more reals than floats, so some corners must be cut. For example:

```
printf "%.20g\n", 123456789123456789;
#        produces 123456789123456784
```

Testing for exact equality of floating-point equality or inequality is not a good idea. Here's a (relatively expensive) work-around to compare whether two floating-point numbers are equal to a particular number of decimal places. See Knuth, volume II, for a more robust treatment of this topic.

```
sub fp_equal {
    my ($X, $Y, $POINTS) = @_;
    my ($tX, $tY);
    $tX = sprintf("%.${POINTS}g", $X);
    $tY = sprintf("%.${POINTS}g", $Y);
    return $tX eq $tY;
}
```

The POSIX module (part of the standard perl distribution) implements *ceil()*, *floor()*, and a number of other mathematical and trigonometric functions. The Math::Complex module (part of the standard perl distribution) defines a number of mathematical functions that can also work on real numbers. Math::Complex not as efficient as POSIX, but POSIX can't work with complex numbers.

Rounding in financial applications can have serious implications, and the rounding method used should be specified precisely. In these cases, it probably pays not to trust whichever system rounding is being used by Perl, but to instead implement the rounding function you need yourself.

**Bigger Numbers**

The standard Math::BigInt and Math::BigFloat modules provide variable precision arithmetic and overloaded operators. At the cost of some space and considerable speed, they avoid the normal pitfalls associated with limited-precision representations.

```
use Math::BigInt;
$x = Math::BigInt->new('123456789123456789');
print $x * $x;

# prints +15241578780673678515622620750190521
```

**NAME**

perlre – Perl regular expressions

**DESCRIPTION**

This page describes the syntax of regular expressions in Perl. For a description of how to *use* regular expressions in matching operations, plus various examples of the same, see discussion of `m//`, `s///`, `qr//` and `??` in the section on *Regexp Quote-Like Operators* in the *perlop* manpage.

The matching operations can have various modifiers. The modifiers that relate to the interpretation of the regular expression inside are listed below. For the modifiers that alter the way a regular expression is used by Perl, see the section on *Regexp Quote-Like Operators* in the *perlop* manpage and the section on *Gory details of parsing quoted constructs* in the *perlop* manpage.

i Do case-insensitive pattern matching.

  If `use locale` is in effect, the case map is taken from the current locale. See the *perllocale* manpage.

m Treat string as multiple lines. That is, change "^" and "$" from matching at only the very start or end of the string to the start or end of any line anywhere within the string,

s Treat string as single line. That is, change "." to match any character whatsoever, even a newline, which it normally would not match.

  The `/s` and `/m` modifiers both override the `$*` setting. That is, no matter what `$*` contains, `/s` without `/m` will force "^" to match only at the beginning of the string and "$" to match only at the end (or just before a newline at the end) of the string. Together, as /ms, they let the "." match any character whatsoever, while yet allowing "^" and "$" to match, respectively, just after and just before newlines within the string.

x Extend your pattern's legibility by permitting whitespace and comments.

These are usually written as "the `/x` modifier", even though the delimiter in question might not actually be a slash. In fact, any of these modifiers may also be embedded within the regular expression itself using the new `(?...)` construct. See below.

The `/x` modifier itself needs a little more explanation. It tells the regular expression parser to ignore whitespace that is neither backslashed nor within a character class. You can use this to break up your regular expression into (slightly) more readable parts. The `#` character is also treated as a metacharacter introducing a comment, just as in ordinary Perl code. This also means that if you want real whitespace or `#` characters in the pattern (outside of a character class, where they are unaffected by `/x`), that you'll either have to escape them or encode them using octal or hex escapes. Taken together, these features go a long way towards making Perl's regular expressions more readable. Note that you have to be careful not to include the pattern delimiter in the comment—perl has no way of knowing you did not intend to close the pattern early. See the C–comment deletion code in the *perlop* manpage.

### Regular Expressions

The patterns used in pattern matching are regular expressions such as those supplied in the Version 8 regex routines. (In fact, the routines are derived (distantly) from Henry Spencer's freely redistributable reimplementation of the V8 routines.) See the section on *Version 8 Regular Expressions* for details.

In particular the following metacharacters have their standard *egrep*–ish meanings:

```
\    Quote the next metacharacter
^    Match the beginning of the line
.    Match any character (except newline)
$    Match the end of the line (or before newline at the end)
|    Alternation
()   Grouping
[]   Character class
```

By default, the "^" character is guaranteed to match at only the beginning of the string, the "$" character at only the end (or before the newline at the end) and Perl does certain optimizations with the assumption that the string contains only one line. Embedded newlines will not be matched by "^" or "$". You may, however, wish to treat a string as a multi-line buffer, such that the "^" will match after any newline within the string, and "$" will match before any newline. At the cost of a little more overhead, you can do this by using the /m modifier on the pattern match operator. (Older programs did this by setting `$*`, but this practice is now deprecated.)

To facilitate multi-line substitutions, the "." character never matches a newline unless you use the `/s` modifier, which in effect tells Perl to pretend the string is a single line—even if it isn't. The `/s` modifier also overrides the setting of `$*`, in case you have some (badly behaved) older code that sets it in another module.

The following standard quantifiers are recognized:

```
*      Match 0 or more times
+      Match 1 or more times
?      Match 1 or 0 times
{n}    Match exactly n times
{n,}   Match at least n times
{n,m}  Match at least n but not more than m times
```

(If a curly bracket occurs in any other context, it is treated as a regular character.) The "*" modifier is equivalent to `{0,}`, the "+" modifier to `{1,}`, and the "?" modifier to `{0,1}`. n and m are limited to integral values less than 65536.

By default, a quantified subpattern is "greedy", that is, it will match as many times as possible (given a particular starting location) while still allowing the rest of the pattern to match. If you want it to match the minimum number of times possible, follow the quantifier with a "?". Note that the meanings don't change, just the "greediness":

```
*?     Match 0 or more times
+?     Match 1 or more times
??     Match 0 or 1 time
{n}?   Match exactly n times
{n,}?  Match at least n times
{n,m}? Match at least n but not more than m times
```

Because patterns are processed as double quoted strings, the following also work:

```
\t          tab                   (HT, TAB)
\n          newline               (LF, NL)
\r          return                (CR)
\f          form feed             (FF)
\a          alarm (bell)          (BEL)
\e          escape (think troff)  (ESC)
\033        octal char (think of a PDP-11)
\x1B        hex char
\c[         control char
\l          lowercase next char (think vi)
\u          uppercase next char (think vi)
\L          lowercase till \E (think vi)
\U          uppercase till \E (think vi)
\E          end case modification (think vi)
\Q          quote (disable) pattern metacharacters till \E
```

If `use locale` is in effect, the case map used by `\l`, `\L`, `\u` and `\U` is taken from the current locale. See the *perllocale* manpage.

You cannot include a literal `$` or `@` within a `\Q` sequence. An unescaped `$` or `@` interpolates the corresponding variable, while escaping will cause the literal string `\$` to be matched. You'll need to write something like `m/\Quser\E\@\Qhost/`.

In addition, Perl defines the following:

```
\w  Match a "word" character (alphanumeric plus "_")
\W  Match a non-word character
\s  Match a whitespace character
\S  Match a non-whitespace character
\d  Match a digit character
\D  Match a non-digit character
```

A `\w` matches a single alphanumeric character, not a whole word. To match a word you'd need to say `\w+`. If `use locale` is in effect, the list of alphabetic characters generated by `\w` is taken from the current locale. See the *perllocale* manpage. You may use `\w`, `\W`, `\s`, `\S`, `\d`, and `\D` within character classes (though not as either end of a range).

Perl defines the following zero-width assertions:

```
\b  Match a word boundary
\B  Match a non-(word boundary)
\A  Match only at beginning of string
\Z  Match only at end of string, or before newline at the end
\z  Match only at end of string
\G  Match only where previous m//g left off (works only with /g)
```

A word boundary (`\b`) is defined as a spot between two characters that has a `\w` on one side of it and a `\W` on the other side of it (in either order), counting the imaginary characters off the beginning and end of the string as matching a `\W`. (Within character classes `\b` represents backspace rather than a word boundary.) The `\A` and `\Z` are just like "^" and "$", except that they won't match multiple times when the `/m` modifier is used, while "^" and "$" will match at every internal line boundary. To match the actual end of the string, not ignoring newline, you can use `\z`. The `\G` assertion can be used to chain global matches (using `m//g`), as described in the section on *Regexp Quote-Like Operators* in the *perlop* manpage.

It is also useful when writing `lex`–like scanners, when you have several patterns that you want to match against consequent substrings of your string, see the previous reference. The actual location where `\G` will match can also be influenced by using `pos()` as an lvalue. See the `pos` entry in the *perlfunc* manpage.

When the bracketing construct `( ... )` is used, \<digit> matches the digit'th substring. Outside of the pattern, always use "$" instead of "\" in front of the digit. (While the \<digit> notation can on rare occasion work outside the current pattern, this should not be relied upon. See the WARNING below.) The scope of $<digit> (and `` $` ``, `$&`, and `$'`) extends to the end of the enclosing BLOCK or eval string, or to the next successful pattern match, whichever comes first. If you want to use parentheses to delimit a subpattern (e.g., a set of alternatives) without saving it as a subpattern, follow the ( with a ?:.

You may have as many parentheses as you wish. If you have more than 9 substrings, the variables `$10`, `$11`, ... refer to the corresponding substring. Within the pattern, \10, \11, etc. refer back to substrings if there have been at least that many left parentheses before the backreference. Otherwise (for backward compatibility) \10 is the same as \010, a backspace, and \11 the same as \011, a tab. And so on. (\1 through \9 are always backreferences.)

`$+` returns whatever the last bracket match matched. `$&` returns the entire matched string. (`$0` used to return the same thing, but not any more.) `` $` `` returns everything before the matched string. `$'` returns everything after the matched string. Examples:

```
s/^([^ ]*) *([^ ]*)/$2 $1/;     # swap first two words
```

```
if (/Time: (..):(..):(..)/) {
    $hours = $1;
    $minutes = $2;
    $seconds = $3;
}
```

Once perl sees that you need one of `$&`, `` $` `` or `$'` anywhere in the program, it has to provide them on each and every pattern match. This can slow your program down. The same mechanism that handles these provides for the use of `$1`, `$2`, etc., so you pay the same price for each pattern that contains capturing parentheses. But if you never use $&, etc., in your script, then patterns *without* capturing parentheses won't be penalized. So avoid $&, $', and $` if you can, but if you can't (and some algorithms really appreciate them), once you've used them once, use them at will, because you've already paid the price. As of 5.005, $& is not so costly as the other two.

Backslashed metacharacters in Perl are alphanumeric, such as `\b`, `\w`, `\n`. Unlike some other regular expression languages, there are no backslashed symbols that aren't alphanumeric. So anything that looks like \\, \(, \), \<, \>, \{, or \} is always interpreted as a literal character, not a metacharacter. This was once used in a common idiom to disable or quote the special meanings of regular expression metacharacters in a string that you want to use for a pattern. Simply quote all non-alphanumeric characters:

```
$pattern =~ s/(\W)/\\$1/g;
```

Now it is much more common to see either the *quotemeta()* function or the `\Q` escape sequence used to disable all metacharacters' special meanings like this:

```
/$unquoted\Q$quoted\E$unquoted/
```

Perl defines a consistent extension syntax for regular expressions. The syntax is a pair of parentheses with a question mark as the first thing within the parentheses (this was a syntax error in older versions of Perl). The character after the question mark gives the function of the extension. Several extensions are already supported:

`(?#text)`

A comment. The text is ignored. If the `/x` switch is used to enable whitespace formatting, a simple `#` will suffice. Note that perl closes the comment as soon as it sees a `)`, so there is no way to put a literal `)` in the comment.

`(?:pattern)`

`(?imsx-imsx:pattern)`

This is for clustering, not capturing; it groups subexpressions like "()", but doesn't make back-references as "()" does. So

```
@fields = split(/\b(?:a|b|c)\b/)
```

is like

```
@fields = split(/\b(a|b|c)\b/)
```

but doesn't spit out extra fields.

The letters between `?` and `:` act as flags modifiers, see the `(?imsx-imsx)` manpage. In particular,

```
/(?s-i:more.*than).*million/i
```

is equivalent to more verbose

```
/(?:(?s-i)more.*than).*million/i
```

`(?=pattern)`
: A zero-width positive lookahead assertion. For example, `/\w+(?=\t)/` matches a word followed by a tab, without including the tab in `$&`.

`(?!pattern)`
: A zero-width negative lookahead assertion. For example `/foo(?!bar)/` matches any occurrence of "foo" that isn't followed by "bar". Note however that lookahead and lookbehind are NOT the same thing. You cannot use this for lookbehind.

If you are looking for a "bar" that isn't preceded by a "foo", `/(?!foo)bar/` will not do what you want. That's because the `(?!foo)` is just saying that the next thing cannot be "foo"—and it's not, it's a "bar", so "foobar" will match. You would have to do something like `/(?!foo)...bar/` for that. We say "like" because there's the case of your "bar" not having three characters before it. You could cover that this way: `/(?:(?!foo)...|^.{0,2})bar/`. Sometimes it's still easier just to say:

```
if (/bar/ && $` !~ /foo$/)
```

For lookbehind see below.

`(?<=pattern)`
: A zero-width positive lookbehind assertion. For example, `/(?<=\t)\w+/` matches a word following a tab, without including the tab in `$&`. Works only for fixed-width lookbehind.

`(?<!pattern)`
: A zero-width negative lookbehind assertion. For example `/(?<!bar)foo/` matches any occurrence of "foo" that isn't following "bar". Works only for fixed-width lookbehind.

`(?{ code })`
: Experimental "evaluate any Perl code" zero-width assertion. Always succeeds. `code` is not interpolated. Currently the rules to determine where the `code` ends are somewhat convoluted.

Owing to the risks to security, this is only available when the `use re 'eval'` pragma is used, and then only for patterns that don't have any variables that must be interpolated at run time.

The `code` is properly scoped in the following sense: if the assertion is backtracked (compare the section on *Backtracking*), all the changes introduced after `local`isation are undone, so

```
$_ = 'a' x 8;
m<
   (?{ $cnt = 0 })                    # Initialize $cnt.
   (
     a
     (?{
         local $cnt = $cnt + 1;       # Update $cnt, backtracking-safe.
     })
   )*
   aaaa
   (?{ $res = $cnt })                 # On success copy to non-localized
                                      # location.
 >x;
```

will set `$res = 4`. Note that after the match `$cnt` returns to the globally introduced value 0, since the scopes which restrict `local` statements are unwound.

This assertion may be used as `(?(condition)yes-pattern` switch. If *not* used in this way, the result of evaluation of `code` is put into variable $^R. This happens immediately, so $^R can be used from other `(?{ code })` assertions inside the same regular expression.

The above assignment to $^R is properly localized, thus the old value of $^R is restored if the assertion is backtracked (compare the section on *Backtracking*).

`(?>pattern)`
An "independent" subexpression. Matches the substring that a *standalone* `pattern` would match if anchored at the given position, **and only this substring**.

Say, `^(?>a*)ab` will never match, since `(?>a*)` (anchored at the beginning of string, as above) will match *all* characters `a` at the beginning of string, leaving no `a` for `ab` to match. In contrast, `a*ab` will match the same as `a+b`, since the match of the subgroup `a*` is influenced by the following group `ab` (see the section on *Backtracking*). In particular, `a*` inside `a*ab` will match less characters that a standalone `a*`, since this makes the tail match.

An effect similar to `(?>pattern)` may be achieved by

```
(?=(pattern))\1
```

since the lookahead is in *"logical"* context, thus matches the same substring as a standalone `a+`. The following `\1` eats the matched string, thus making a zero-length assertion into an analogue of `(?...)>`. (The difference between these two constructs is that the second one uses a catching group, thus shifting ordinals of backreferences in the rest of a regular expression.)

This construct is useful for optimizations of "eternal" matches, because it will not backtrack (see the section on *Backtracking*).

```
m{   \( (
        [^()]+
      |
        \( [^()]* \)
      )+
       \)
 }x
```

That will efficiently match a nonempty group with matching two-or-less-level-deep parentheses. However, if there is no such group, it will take virtually forever on a long string. That's because there are so many different ways to split a long string into several substrings. This is essentially what `(.+)+` is doing, and this is a subpattern of the above pattern. Consider that `((()aaaaaaaaaaaaaaaaaa` on the pattern above detects no-match in several seconds, but that each extra letter doubles this time. This exponential performance will make it appear that your program has hung.

However, a tiny modification of this pattern

```
m{ \( (
        (?> [^()]+ )
      |
        \( [^()]* \)
      )+
       \)
 }x
```

which uses `(?>...)` matches exactly when the one above does (verifying this yourself would be a productive exercise), but finishes in a fourth the time when used on a similar string with 1000000 `a`s. Be aware, however, that this pattern currently triggers a warning message under **−w** saying it `"matches the null string many times"`):

On simple groups, such as the pattern `(? [^()]+ )>`, a comparable effect may be achieved by negative lookahead, as in `[^()]+ (?! [^()] )`. This was only 4 times slower on a string with 1000000 `a`s.

`(?(condition)yes-pattern|no-pattern)`

`(?(condition)yes-pattern)`

Conditional expression. `(condition)` should be either an integer in parentheses (which is valid if the corresponding pair of parentheses matched), or lookahead/lookbehind/evaluate zero-width assertion.

Say,

```
m{ ( \( )?
   [^()]+
    (?(1) \) )
}x
```

matches a chunk of non-parentheses, possibly included in parentheses themselves.

`(?imsx-imsx)`

One or more embedded pattern-match modifiers. This is particularly useful for patterns that are specified in a table somewhere, some of which want to be case sensitive, and some of which don't. The case insensitive ones need to include merely `(?i)` at the front of the pattern. For example:

```
$pattern = "foobar";
if ( /$pattern/i ) { }

# more flexible:

$pattern = "(?i)foobar";
if ( /$pattern/ ) { }
```

Letters after − switch modifiers off.

These modifiers are localized inside an enclosing group (if any). Say,

```
( (?i) blah ) \s+ \1
```

(assuming `x` modifier, and no `i` modifier outside of this group) will match a repeated (*including the case*!) word `blah` in any case.

A question mark was chosen for this and for the new minimal-matching construct because 1) question mark is pretty rare in older regular expressions, and 2) whenever you see one, you should stop and "question" exactly what is going on. That's psychology...

**Backtracking**

A fundamental feature of regular expression matching involves the notion called *backtracking*, which is currently used (when needed) by all regular expression quantifiers, namely `*`, `*?`, `+`, `+?`, `{n,m}`, and `{n,m}?`.

For a regular expression to match, the *entire* regular expression must match, not just part of it. So if the beginning of a pattern containing a quantifier succeeds in a way that causes later parts in the pattern to fail, the matching engine backs up and recalculates the beginning part—that's why it's called backtracking.

Here is an example of backtracking: Let's say you want to find the word following "foo" in the string "Food is on the foo table.":

```
$_ = "Food is on the foo table.";
if ( /\b(foo)\s+(\w+)/i ) {
    print "$2 follows $1.\n";
}
```

When the match runs, the first part of the regular expression (`\b(foo)`) finds a possible match right at the beginning of the string, and loads up `$1` with "Foo". However, as soon as the matching engine sees that there's no whitespace following the "Foo" that it had saved in `$1`, it realizes its mistake and starts over again one character after where it had the tentative match. This time it goes all the way until the next occurrence of "foo". The complete regular expression matches this time, and you get the expected output of "table follows foo."

Sometimes minimal matching can help a lot. Imagine you'd like to match everything between "foo" and "bar". Initially, you write something like this:

```
    $_ =  "The food is under the bar in the barn.";
    if ( /foo(.*)bar/ ) {
        print "got <$1>\n";
    }
```

Which perhaps unexpectedly yields:

```
  got <d is under the bar in the >
```

That's because `.*` was greedy, so you get everything between the *first* "foo" and the *last* "bar". In this case, it's more effective to use minimal matching to make sure you get the text between a "foo" and the first "bar" thereafter.

```
    if ( /foo(.*?)bar/ ) { print "got <$1>\n" }
  got <d is under the >
```

Here's another example: let's say you'd like to match a number at the end of a string, and you also want to keep the preceding part the match. So you write this:

```
    $_ = "I have 2 numbers: 53147";
    if ( /(.*)(\d*)/ ) {                                # Wrong!
        print "Beginning is <$1>, number is <$2>.\n";
    }
```

That won't work at all, because `.*` was greedy and gobbled up the whole string. As `\d*` can match on an empty string the complete regular expression matched successfully.

```
    Beginning is <I have 2 numbers: 53147>, number is <>.
```

Here are some variants, most of which don't work:

```
    $_ = "I have 2 numbers: 53147";
    @pats = qw{
        (.*)(\d*)
        (.*)(\d+)
        (.*?)(\d*)
        (.*?)(\d+)
        (.*)(\d+)$
        (.*?)(\d+)$
        (.*)\b(\d+)$
        (.*\D)(\d+)$
    };
```

```
for $pat (@pats) {
    printf "%-12s ", $pat;
    if ( /$pat/ ) {
        print "<$1> <$2>\n";
    } else {
        print "FAIL\n";
    }
}
```

That will print out:

```
(.*)(\d*)     <I have 2 numbers: 53147> <>
(.*)(\d+)     <I have 2 numbers: 5314> <7>
(.*?)(\d*)    <> <>
(.*?)(\d+)    <I have > <2>
(.*)(\d+)$    <I have 2 numbers: 5314> <7>
(.*?)(\d+)$   <I have 2 numbers: > <53147>
(.*)\b(\d+)$  <I have 2 numbers: > <53147>
(.*\D)(\d+)$  <I have 2 numbers: > <53147>
```

As you see, this can be a bit tricky. It's important to realize that a regular expression is merely a set of assertions that gives a definition of success. There may be 0, 1, or several different ways that the definition might succeed against a particular string. And if there are multiple ways it might succeed, you need to understand backtracking to know which variety of success you will achieve.

When using lookahead assertions and negations, this can all get even tricker. Imagine you'd like to find a sequence of non-digits not followed by "123". You might try to write that as

```
    $_ = "ABC123";
    if ( /^\D*(?!123)/ ) {                          # Wrong!
        print "Yup, no 123 in $_\n";
    }
```

But that isn't going to match; at least, not the way you're hoping. It claims that there is no 123 in the string. Here's a clearer picture of why it that pattern matches, contrary to popular expectations:

```
$x = 'ABC123' ;
$y = 'ABC445' ;

print "1: got $1\n" if $x =~ /^(ABC)(?!123)/ ;
print "2: got $1\n" if $y =~ /^(ABC)(?!123)/ ;

print "3: got $1\n" if $x =~ /^(\D*)(?!123)/ ;
print "4: got $1\n" if $y =~ /^(\D*)(?!123)/ ;
```

This prints

```
2: got ABC
3: got AB
4: got ABC
```

You might have expected test 3 to fail because it seems to a more general purpose version of test 1. The important difference between them is that test 3 contains a quantifier (`\D*`) and so can use backtracking, whereas test 1 will not. What's happening is that you've asked "Is it true that at the start of `$x`, following 0 or more non-digits, you have something that's not 123?" If the pattern matcher had let `\D*` expand to "ABC", this would have caused the whole pattern to fail. The search engine will initially match `\D*` with "ABC". Then it will try to match `(?!123` with "123", which of course fails. But because a quantifier

(`\D*`) has been used in the regular expression, the search engine can backtrack and retry the match differently in the hope of matching the complete regular expression.

The pattern really, *really* wants to succeed, so it uses the standard pattern back-off-and-retry and lets `\D*` expand to just "AB" this time. Now there's indeed something following "AB" that is not "123". It's in fact "C123", which suffices.

We can deal with this by using both an assertion and a negation. We'll say that the first part in `$1` must be followed by a digit, and in fact, it must also be followed by something that's not "123". Remember that the lookaheads are zero-width expressions— they only look, but don't consume any of the string in their match. So rewriting this way produces what you'd expect; that is, case 5 will fail, but case 6 succeeds:

```
print "5: got $1\n" if $x =~ /^(\D*)(?=\d)(?!123)/ ;
print "6: got $1\n" if $y =~ /^(\D*)(?=\d)(?!123)/ ;

6: got ABC
```

In other words, the two zero-width assertions next to each other work as though they're ANDed together, just as you'd use any builtin assertions: `/^$/` matches only if you're at the beginning of the line AND the end of the line simultaneously. The deeper underlying truth is that juxtaposition in regular expressions always means AND, except when you write an explicit OR using the vertical bar. `/ab/` means match "a" AND (then) match "b", although the attempted matches are made at different positions because "a" is not a zero-width assertion, but a one-width assertion.

One warning: particularly complicated regular expressions can take exponential time to solve due to the immense number of possible ways they can use backtracking to try match. For example this will take a very long time to run

```
/((a{0,5}){0,5}){0,5}/
```

And if you used `*`'s instead of limiting it to 0 through 5 matches, then it would take literally forever— or until you ran out of stack space.

A powerful tool for optimizing such beasts is "independent" groups, which do not backtrace (see the `(?>pattern)` manpage). Note also that zero-length lookahead/lookbehind assertions will not backtrace to make the tail match, since they are in "logical" context: only the fact whether they match or not is considered relevant. For an example where side-effects of a lookahead *might* have influenced the following match, see the `(?>pattern)` manpage.

**Version 8 Regular Expressions**

In case you're not familiar with the "regular" Version 8 regex routines, here are the pattern-matching rules not described above.

Any single character matches itself, unless it is a *metacharacter* with a special meaning described here or above. You can cause characters that normally function as metacharacters to be interpreted literally by prefixing them with a "\" (e.g., "\." matches a ".", not any character; "\\" matches a "\"). A series of characters matches that series of characters in the target string, so the pattern `blurfl` would match "blurfl" in the target string.

You can specify a character class, by enclosing a list of characters in `[]`, which will match any one character from the list. If the first character after the "[" is "^", the class matches any character not in the list. Within a list, the "−" character is used to specify a range, so that `a-z` represents all characters between "a" and "z", inclusive. If you want "−" itself to be a member of a class, put it at the start or end of the list, or escape it with a backslash. (The following all specify the same class of three characters: `[-az]`, `[az-]`, and `[a\-z]`. All are different from `[a-z]`, which specifies a class containing twenty-six characters.)

Characters may be specified using a metacharacter syntax much like that used in C: "\n" matches a newline, "\t" a tab, "\r" a carriage return, "\f" a form feed, etc. More generally, *nnn*, where *nnn* is a string of octal digits, matches the character whose ASCII value is *nnn*. Similarly, \x*nn*, where *nn* are hexadecimal digits, matches the character whose ASCII value is *nn*. The expression \c*x* matches the ASCII character control-*x*. Finally, the "." metacharacter matches any character except "\n" (unless you use `/s`).

You can specify a series of alternatives for a pattern using "|" to separate them, so that `fee|fie|foe` will match any of "fee", "fie", or "foe" in the target string (as would `f(e|i|o)e`). The first alternative includes everything from the last pattern delimiter ("(", "[", or the beginning of the pattern) up to the first "|", and the last alternative contains everything from the last "|" to the next pattern delimiter. For this reason, it's common practice to include alternatives in parentheses, to minimize confusion about where they start and end.

Alternatives are tried from left to right, so the first alternative found for which the entire expression matches, is the one that is chosen. This means that alternatives are not necessarily greedy. For example: when mathing `foo|foot` against "barefoot", only the "foo" part will match, as that is the first alternative tried, and it successfully matches the target string. (This might not seem important, but it is important when you are capturing matched text using parentheses.)

Also remember that "|" is interpreted as a literal within square brackets, so if you write `[fee|fie|foe]` you're really only matching `[feio|]`.

Within a pattern, you may designate subpatterns for later reference by enclosing them in parentheses, and you may refer back to the *n*th subpattern later in the pattern using the metacharacter *n*. Subpatterns are numbered based on the left to right order of their opening parenthesis. A backreference matches whatever actually matched the subpattern in the string being examined, not the rules for that subpattern. Therefore, `(0|0x)\d*\s\1\d*` will match "0x1234 0x4321", but not "0x1234 01234", because subpattern 1 actually matched "0x", even though the rule `0|0x` could potentially match the leading 0 in the second number.

### WARNING on \1 vs `$1`

Some people get too used to writing things like:

```
$pattern =~ s/(\W)/\\\1/g;
```

This is grandfathered for the RHS of a substitute to avoid shocking the **sed** addicts, but it's a dirty habit to get into. That's because in PerlThink, the righthand side of a `s///` is a double-quoted string. `\1` in the usual double-quoted string means a control-A. The customary Unix meaning of `\1` is kludged in for `s///`. However, if you get into the habit of doing that, you get yourself into trouble if you then add an `/e` modifier.

```
s/(\d+)/ \1 + 1 /eg;         # causes warning under -w
```

Or if you try to do

```
s/(\d+)/\1000/;
```

You can't disambiguate that by saying `\{1}000`, whereas you can fix it with `${1}000`. Basically, the operation of interpolation should not be confused with the operation of matching a backreference. Certainly they mean two different things on the *left* side of the `s///`.

### Repeated patterns matching zero-length substring

WARNING: Difficult material (and prose) ahead. This section needs a rewrite.

Regular expressions provide a terse and powerful programming language. As with most other power tools, power comes together with the ability to wreak havoc.

A common abuse of this power stems from the ability to make infinite loops using regular expressions, with something as innocous as:

```
'foo' =~ m{ ( o? )* }x;
```

The `o?` can match at the beginning of `'foo'`, and since the position in the string is not moved by the match, `o?` would match again and again due to the `*` modifier. Another common way to create a similar cycle is with the looping modifier `//g`:

```
@matches = ( 'foo' =~ m{ o? }xg );
```

or

```
print "match: <$&>\n" while 'foo' =~ m{ o? }xg;
```

or the loop implied by *split()*.

However, long experience has shown that many programming tasks may be significantly simplified by using repeated subexpressions which may match zero-length substrings, with a simple example being:

```
@chars = split //, $string;           # // is not magic in split
($whitewashed = $string) =~ s/()/ /g; # parens avoid magic s// /
```

Thus Perl allows the `/()/` construct, which *forcefully breaks the infinite loop*. The rules for this are different for lower-level loops given by the greedy modifiers `*+{}`, and for higher-level ones like the `/g` modifier or *split()* operator.

The lower-level loops are *interrupted* when it is detected that a repeated expression did match a zero-length substring, thus

```
m{ (?: NON_ZERO_LENGTH | ZERO_LENGTH )* }x;
```

is made equivalent to

```
m{   (?: NON_ZERO_LENGTH )*
   |
     (?: ZERO_LENGTH )?
 }x;
```

The higher level-loops preserve an additional state between iterations: whether the last match was zero-length. To break the loop, the following match after a zero-length match is prohibited to have a length of zero. This prohibition interacts with backtracking (see the section on *Backtracking*), and so the *second best* match is chosen if the *best* match is of zero length.

Say,

```
$_ = 'bar';
s/\w??/<$&>/g;
```

results in "<<b><><a><><r><>". At each position of the string the best match given by non-greedy `??` is the zero-length match, and the *second best* match is what is matched by `\w`. Thus zero-length matches alternate with one-character-long matches.

Similarly, for repeated `m/()/g` the second-best match is the match at the position one notch further in the string.

The additional state of being *matched with zero-length* is associated to the matched string, and is reset by each assignment to *pos()*.

**Creating custom RE engines**

Overloaded constants (see the *overload* manpage) provide a simple way to extend the functionality of the RE engine.

Suppose that we want to enable a new RE escape-sequence `\Y|` which matches at boundary between white-space characters and non-whitespace characters. Note that `(?=\S)(?<!\S)|(?!\S)(?<=\S)` matches exactly at these positions, so we want to have each `\Y|` in the place of the more complicated version. We can create a module `customre` to do this:

```
package customre;
use overload;

sub import {
  shift;
  die "No argument to customre::import allowed" if @_;
  overload::constant 'qr' => \&convert;
}

sub invalid { die "/$_[0]/: invalid escape '\\$_[1]'"}

my %rules = ( '\\' => '\\',
              'Y|' => qr/(?=\S)(?<!\S)|(?!\S)(?<=\S)/ );
sub convert {
  my $re = shift;
  $re =~ s{
            \\ ( \\ | Y . )
          }
          { $rules{$1} or invalid($re,$1) }sgex;
  return $re;
}
```

Now `use customre` enables the new escape in constant regular expressions, i.e., those without any run-time variable interpolations. As documented in the *overload* manpage, this conversion will work only over literal parts of regular expressions. For `\Y|$re\Y|` the variable part of this regular expression needs to be converted explicitly (but only if the special meaning of `\Y|` should be enabled inside `$re`):

```
use customre;
$re = <>;
chomp $re;
$re = customre::convert $re;
/\Y|$re\Y|/;
```

**SEE ALSO**

the section on *Regexp Quote-Like Operators* in the *perlop* manpage.

the section on *Gory details of parsing quoted constructs* in the *perlop* manpage.

the `pos` entry in the *perlfunc* manpage.

the *perllocale* manpage.

*Mastering Regular Expressions* (see the *perlbook* manpage) by Jeffrey Friedl.

**NAME**

perlrun − how to execute the Perl interpreter

**SYNOPSIS**

**perl** [ **−sTuU** ] [ **−hv** ] [ **−V**[:*configvar*] ]
[ **−cw** ] [ **−d**[:*debugger*] ] [ **−D**[*number/list*] ]
[ **−pna** ] [ **−F***pattern* ] [ **−l**[*octal*] ] [ **−0**[*octal*] ]
[ **−I***dir* ] [ **−m**[**−**]*module* ] [ **−M**[**−**]*'module...'* ] [ **−P** ] [ **−S** ] [ **−x**[*dir*] ]
[ **−i**[*extension*] ] [ **−e** *'command'* ] [ **--** ] [ *programfile* ] [ *argument* ]...

**DESCRIPTION**

Upon startup, Perl looks for your script in one of the following places:

1. Specified line by line via **−e** switches on the command line.
2. Contained in the file specified by the first filename on the command line. (Note that systems supporting the #! notation invoke interpreters this way. See the section on *Location of Perl*.)
3. Passed in implicitly via standard input. This works only if there are no filename arguments— to pass arguments to a STDIN script you must explicitly specify a "−" for the script name.

With methods 2 and 3, Perl starts parsing the input file from the beginning, unless you've specified a **−x** switch, in which case it scans for the first line starting with #! and containing the word "perl", and starts there instead. This is useful for running a script embedded in a larger message. (In this case you would indicate the end of the script using the `__END__` token.)

The #! line is always examined for switches as the line is being parsed. Thus, if you're on a machine that allows only one argument with the #! line, or worse, doesn't even recognize the #! line, you still can get consistent switch behavior regardless of how Perl was invoked, even if **−x** was used to find the beginning of the script.

Because many operating systems silently chop off kernel interpretation of the #! line after 32 characters, some switches may be passed in on the command line, and some may not; you could even get a "−" without its letter, if you're not careful. You probably want to make sure that all your switches fall either before or after that 32 character boundary. Most switches don't actually care if they're processed redundantly, but getting a − instead of a complete switch could cause Perl to try to execute standard input instead of your script. And a partial **−I** switch could also cause odd results.

Some switches do care if they are processed twice, for instance combinations of **−l** and **−0**. Either put all the switches after the 32 character boundary (if applicable), or replace the use of **−0***digits* by `BEGIN{ $/ = "\0digits"; }`.

Parsing of the #! switches starts wherever "perl" is mentioned in the line. The sequences "−*" and "− " are specifically ignored so that you could, if you were so inclined, say

```
#!/bin/sh -- # -*- perl -*- -p
eval 'exec /usr/bin/perl -wS $0 ${1+"$@"}'
    if $running_under_some_shell;
```

to let Perl see the **−p** switch.

If the #! line does not contain the word "perl", the program named after the #! is executed instead of the Perl interpreter. This is slightly bizarre, but it helps people on machines that don't do #!, because they can tell a program that their SHELL is /usr/bin/perl, and Perl will then dispatch the program to the correct interpreter for them.

After locating your script, Perl compiles the entire script to an internal form. If there are any compilation errors, execution of the script is not attempted. (This is unlike the typical shell script, which might run partway through before finding a syntax error.)

If the script is syntactically correct, it is executed. If the script runs off the end without hitting an *exit()* or *die()* operator, an implicit `exit(0)` is provided to indicate successful completion.

### #! and quoting on non-Unix systems

Unix's #! technique can be simulated on other systems:

OS/2
: Put

```
extproc perl -S -your_switches
```

as the first line in `*.cmd` file (`-S` due to a bug in cmd.exe's 'extproc' handling).

MS−DOS
: Create a batch file to run your script, and codify it in `ALTERNATIVE_SHEBANG` (see the *dosish.h* file in the source distribution for more information).

Win95/NT
: The Win95/NT installation, when using the Activeware port of Perl, will modify the Registry to associate the *.pl* extension with the perl interpreter. If you install another port of Perl, including the one in the Win32 directory of the Perl distribution, then you'll have to modify the Registry yourself. Note that this means you can no longer tell the difference between an executable Perl program and a Perl library file.

Macintosh
: Macintosh perl scripts will have the appropriate Creator and Type, so that double-clicking them will invoke the perl application.

Command-interpreters on non-Unix systems have rather different ideas on quoting than Unix shells. You'll need to learn the special characters in your command-interpreter (`*`, `\` and `"` are common) and how to protect whitespace and these characters to run one-liners (see `-e` below).

On some systems, you may have to change single-quotes to double ones, which you must *NOT* do on Unix or Plan9 systems. You might also have to change a single % to a %%.

For example:

```
# Unix
perl -e 'print "Hello world\n"'

# MS-DOS, etc.
perl -e "print \"Hello world\n\""

# Macintosh
print "Hello world\n"
 (then Run "Myscript" or Shift-Command-R)

# VMS
perl -e "print ""Hello world\n"""
```

The problem is that none of this is reliable: it depends on the command and it is entirely possible neither works. If 4DOS was the command shell, this would probably work better:

```
perl -e "print <Ctrl-x>"Hello world\n<Ctrl-x>""
```

CMD.EXE in Windows NT slipped a lot of standard Unix functionality in when nobody was looking, but just try to find documentation for its quoting rules.

Under the Macintosh, it depends which environment you are using. The MacPerl shell, or MPW, is much like Unix shells in its support for several quoting variants, except that it makes free use of the Macintosh's non-ASCII characters as control characters.

There is no general solution to all of this. It's just a mess.

### Location of Perl

It may seem obvious to say, but Perl is useful only when users can easily find it. When possible, it's good for both **/usr/bin/perl** and **/usr/local/bin/perl** to be symlinks to the actual binary. If that can't be done, system administrators are strongly encouraged to put (symlinks to) perl and its accompanying utilities, such as perldoc, into a directory typically found along a user's PATH, or in another obvious and convenient place.

In this documentation, `#!/usr/bin/perl` on the first line of the script will stand in for whatever method works on your system.

### Switches

A single-character switch may be combined with the following switch, if any.

```
#!/usr/bin/perl -spi.bak    # same as -s -p -i.bak
```

Switches include:

**–0**[*digits*]
specifies the input record separator (`$/`) as an octal number. If there are no digits, the null character is the separator. Other switches may precede or follow the digits. For example, if you have a version of **find** which can print filenames terminated by the null character, you can say this:

```
find . -name '*.bak' -print0 | perl -n0e unlink
```

The special value 00 will cause Perl to slurp files in paragraph mode. The value 0777 will cause Perl to slurp files whole because there is no legal character with that value.

**–a** turns on autosplit mode when used with a **–n** or **–p**. An implicit split command to the `@F` array is done as the first thing inside the implicit while loop produced by the **–n** or **–p**.

```
perl -ane 'print pop(@F), "\n";'
```

is equivalent to

```
while (<>) {
    @F = split(' ');
    print pop(@F), "\n";
}
```

An alternate delimiter may be specified using **–F**.

**–c** causes Perl to check the syntax of the script and then exit without executing it. Actually, it *will* execute `BEGIN`, `END`, and `use` blocks, because these are considered as occurring outside the execution of your program.

**–d** runs the script under the Perl debugger. See the *perldebug* manpage.

**–d:***foo*
runs the script under the control of a debugging or tracing module installed as Devel::foo. E.g., **–d:DProf** executes the script using the Devel::DProf profiler. See the *perldebug* manpage.

**–D***letters*

**–D***number*
sets debugging flags. To watch how it executes your script, use **–Dtls**. (This works only if debugging is compiled into your Perl.) Another nice value is **–Dx**, which lists your compiled syntax tree. And **–Dr** displays compiled regular expressions. As an alternative, specify a number instead of list of letters (e.g., **–D14** is equivalent to **–Dtls**):

```
      1  p  Tokenizing and parsing
      2  s  Stack snapshots
      4  l  Context (loop) stack processing
      8  t  Trace execution
     16  o  Method and overloading resolution
     32  c  String/numeric conversions
     64  P  Print preprocessor command for -P
    128  m  Memory allocation
    256  f  Format processing
    512  r  Regular expression parsing and execution
   1024  x  Syntax tree dump
   2048  u  Tainting checks
   4096  L  Memory leaks (needs C<-DLEAKTEST> when compiling Perl)
   8192  H  Hash dump -- usurps values()
  16384  X  Scratchpad allocation
  32768  D  Cleaning up
  65536  S  Thread synchronization
```

All these flags require `-DDEBUGGING` when you compile the Perl executable. This flag is automatically set if you include `-g` option when `Configure` asks you about optimizer/debugger flags.

**–e** *commandline*
may be used to enter one line of script. If **–e** is given, Perl will not look for a script filename in the argument list. Multiple **–e** commands may be given to build up a multi-line script. Make sure to use semicolons where you would in a normal program.

**–F***pattern*
specifies the pattern to split on if **–a** is also in effect. The pattern may be surrounded by `//`, `""`, or `''`, otherwise it will be put in single quotes.

**–h** prints a summary of the options.

**–i**[*extension*]
specifies that files processed by the <> construct are to be edited in-place. It does this by renaming the input file, opening the output file by the original name, and selecting that output file as the default for *print()* statements. The extension, if supplied, is used to modify the name of the old file to make a backup copy, following these rules:

If no extension is supplied, no backup is made and the current file is overwritten.

If the extension doesn't contain a `*` then it is appended to the end of the current filename as a suffix.

If the extension does contain one or more `*` characters, then each `*` is replaced with the current filename. In perl terms you could think of this as:

```
($backup = $extension) =~ s/\*/$file_name/g;
```

This allows you to add a prefix to the backup file, instead of (or in addition to) a suffix:

```
$ perl -pi'bak_*' -e 's/bar/baz/' fileA     # backup to 'bak_fileA'
```

Or even to place backup copies of the original files into another directory (provided the directory already exists):

```
$ perl -pi'old/*.bak' -e 's/bar/baz/' fileA # backup to 'old/fileA.bak'
```

These sets of one-liners are equivalent:

```
$ perl -pi -e 's/bar/baz/' fileA            # overwrite current file
$ perl -pi'*' -e 's/bar/baz/' fileA         # overwrite current file

$ perl -pi'.bak' -e 's/bar/baz/' fileA      # backup to 'fileA.bak'
$ perl -pi'*.bak' -e 's/bar/baz/' fileA     # backup to 'fileA.bak'
```

From the shell, saying

```
$ perl -p -i.bak -e "s/foo/bar/; ... "
```

is the same as using the script:

```
#!/usr/bin/perl -pi.bak
s/foo/bar/;
```

which is equivalent to

```
#!/usr/bin/perl
$extension = '.bak';
while (<>) {
    if ($ARGV ne $oldargv) {
        if ($extension !~ /\*/) {
            $backup = $ARGV . $extension;
        }
        else {
            ($backup = $extension) =~ s/\*/$ARGV/g;
        }
        rename($ARGV, $backup);
        open(ARGVOUT, ">$ARGV");
        select(ARGVOUT);
        $oldargv = $ARGV;
    }
    s/foo/bar/;
}
continue {
    print;  # this prints to original filename
}
select(STDOUT);
```

except that the **–i** form doesn't need to compare `$ARGV` to `$oldargv` to know when the filename has changed. It does, however, use ARGVOUT for the selected filehandle. Note that STDOUT is restored as the default output filehandle after the loop.

As shown above, Perl creates the backup file whether or not any output is actually changed. So this is just a fancy way to copy files:

```
    $ perl -p -i'/some/file/path/*' -e 1 file1 file2 file3...
  or
    $ perl -p -i'.bak' -e 1 file1 file2 file3...
```

You can use `eof` without parentheses to locate the end of each input file, in case you want to append to each file, or reset line numbering (see example in the `eof` entry in the *perlfunc* manpage).

If, for a given file, Perl is unable to create the backup file as specified in the extension then it will skip that file and continue on with the next one (if it exists).

For a discussion of issues surrounding file permissions and `-i`, see the section on *Why does Perl let me delete read-only files? Why does -i clobber protected files? Isn't this a bug in Perl?* in the *perlfaq5* manpage.

You cannot use **–i** to create directories or to strip extensions from files.

Perl does not expand `~`, so don't do that.

Finally, note that the **–i** switch does not impede execution when no files are given on the command line. In this case, no backup is made (the original file cannot, of course, be determined) and processing proceeds from STDIN to STDOUT as might be expected.

**–I***directory*
Directories specified by **–I** are prepended to the search path for modules (`@INC`), and also tells the C preprocessor where to search for include files. The C preprocessor is invoked with **–P**; by default it searches /usr/include and /usr/lib/perl.

**–l**[*octnum*]
enables automatic line-ending processing. It has two effects: first, it automatically chomps "`$/`" (the input record separator) when used with **–n** or **–p**, and second, it assigns "`$\`" (the output record separator) to have the value of *octnum* so that any print statements will have that separator added back on. If *octnum* is omitted, sets "`$\`" to the current value of "`$/`". For instance, to trim lines to 80 columns:

```
perl -lpe 'substr($_, 80) = ""'
```

Note that the assignment `$\ = $/` is done when the switch is processed, so the input record separator can be different than the output record separator if the **–l** switch is followed by a **–0** switch:

```
gnufind / -print0 | perl -ln0e 'print "found $_" if -p'
```

This sets `$\` to newline and then sets `$/` to the null character.

**–m**[**–**]*module*

**–M**[**–**]*module*

**–M**[**–**]*'module ...'*

**–[mM]**[**–**]*module=arg[,arg]...*
`-m`*module* executes `use` *module* `();` before executing your script.

`-M`*module* executes `use` *module* `;` before executing your script. You can use quotes to add extra code after the module name, e.g., `-M'module qw(foo bar)'`.

If the first character after the `-M` or `-m` is a dash (`-`) then the 'use' is replaced with 'no'.

A little builtin syntactic sugar means you can also say `-mmodule=foo,bar` or `-Mmodule=foo,bar` as a shortcut for `-M'module qw(foo bar)'`. This avoids the need to use quotes when importing symbols. The actual code generated by `-Mmodule=foo,bar` is `use module split(/,/,q{foo,bar})`. Note that the `=` form removes the distinction between `-m` and `-M`.

**–n** causes Perl to assume the following loop around your script, which makes it iterate over filename arguments somewhat like **sed –n** or **awk**:

```
while (<>) {
    ...             # your script goes here
}
```

Note that the lines are not printed by default. See **–p** to have lines printed. If a file named by an argument cannot be opened for some reason, Perl warns you about it, and moves on to the next file.

Here is an efficient way to delete all files older than a week:

```
find . -mtime +7 -print | perl -nle 'unlink;'
```

This is faster than using the `-exec` switch of **find** because you don't have to start a process on every filename found.

`BEGIN` and `END` blocks may be used to capture control before or after the implicit loop, just as in **awk**.

**-p** causes Perl to assume the following loop around your script, which makes it iterate over filename arguments somewhat like **sed**:

```
while (<>) {
    ...             # your script goes here
} continue {
    print or die "-p destination: $!\n";
}
```

If a file named by an argument cannot be opened for some reason, Perl warns you about it, and moves on to the next file. Note that the lines are printed automatically. An error occuring during printing is treated as fatal. To suppress printing use the **-n** switch. A **-p** overrides a **-n** switch.

`BEGIN` and `END` blocks may be used to capture control before or after the implicit loop, just as in awk.

**-P** causes your script to be run through the C preprocessor before compilation by Perl. (Because both comments and cpp directives begin with the # character, you should avoid starting comments with any words recognized by the C preprocessor such as "if", "else", or "define".)

**-s** enables some rudimentary switch parsing for switches on the command line after the script name but before any filename arguments (or before a **--**). Any switch found there is removed from `@ARGV` and sets the corresponding variable in the Perl script. The following script prints "true" if and only if the script is invoked with a **-xyz** switch.

```
#!/usr/bin/perl -s
if ($xyz) { print "true\n"; }
```

**-S** makes Perl use the PATH environment variable to search for the script (unless the name of the script contains directory separators). On some platforms, this also makes Perl append suffixes to the filename while searching for it. For example, on Win32 platforms, the ".bat" and ".cmd" suffixes are appended if a lookup for the original name fails, and if the name does not already end in one of those suffixes. If your Perl was compiled with DEBUGGING turned on, using the -Dp switch to Perl shows how the search progresses.

If the filename supplied contains directory separators (i.e. it is an absolute or relative pathname), and if the file is not found, platforms that append file extensions will do so and try to look for the file with those extensions added, one by one.

On DOS-like platforms, if the script does not contain directory separators, it will first be searched for in the current directory before being searched for on the PATH. On Unix platforms, the script will be searched for strictly on the PATH.

Typically this is used to emulate #! startup on platforms that don't support #!. This example works on many platforms that have a shell compatible with Bourne shell:

```
#!/usr/bin/perl
eval 'exec /usr/bin/perl -wS $0 ${1+"$@"}'
        if $running_under_some_shell;
```

The system ignores the first line and feeds the script to /bin/sh, which proceeds to try to execute the Perl script as a shell script. The shell executes the second line as a normal shell command, and thus starts up the Perl interpreter. On some systems `$0` doesn't always contain the full pathname, so the **–S** tells Perl to search for the script if necessary. After Perl locates the script, it parses the lines and ignores them because the variable `$running_under_some_shell` is never true. If the script will be interpreted by csh, you will need to replace `${1+"$@"}` with `$*`, even though that doesn't understand embedded spaces (and such) in the argument list. To start up sh rather than csh, some systems may have to replace the #! line with a line containing just a colon, which will be politely ignored by Perl. Other systems can't control that, and need a totally devious construct that will work under any of csh, sh, or Perl, such as the following:

```
eval '(exit $?0)' && eval 'exec /usr/bin/perl -wS $0 ${1+"$@"}'
& eval 'exec /usr/bin/perl -wS $0 $argv:q'
        if $running_under_some_shell;
```

**–T** forces "taint" checks to be turned on so you can test them. Ordinarily these checks are done only when running setuid or setgid. It's a good idea to turn them on explicitly for programs run on another's behalf, such as CGI programs. See the *perlsec* manpage. Note that (for security reasons) this option must be seen by Perl quite early; usually this means it must appear early on the command line or in the #! line (for systems which support that).

**–u** causes Perl to dump core after compiling your script. You can then in theory take this core dump and turn it into an executable file by using the **undump** program (not supplied). This speeds startup at the expense of some disk space (which you can minimize by stripping the executable). (Still, a "hello world" executable comes out to about 200K on my machine.) If you want to execute a portion of your script before dumping, use the *dump()* operator instead. Note: availability of **undump** is platform specific and may not be available for a specific port of Perl. It has been superseded by the new perl-to-C compiler, which is more portable, even though it's still only considered beta.

**–U** allows Perl to do unsafe operations. Currently the only "unsafe" operations are the unlinking of directories while running as superuser, and running setuid programs with fatal taint checks turned into warnings. Note that the **–w** switch (or the `$^W` variable) must be used along with this option to actually **generate** the taint-check warnings.

**–v** prints the version and patchlevel of your Perl executable.

**–V** prints summary of the major perl configuration values and the current value of `@INC`.

**–V:***name*

Prints to STDOUT the value of the named configuration variable.

**–w** prints warnings about variable names that are mentioned only once, and scalar variables that are used before being set. Also warns about redefined subroutines, and references to undefined filehandles or filehandles opened read-only that you are attempting to write on. Also warns you if you use values as a number that doesn't look like numbers, using an array as though it were a scalar, if your subroutines recurse more than 100 deep, and innumerable other things.

You can disable specific warnings using `__WARN__` hooks, as described in the *perlvar* manpage and the `warn` entry in the *perlfunc* manpage. See also the *perldiag* manpage and the *perltrap* manpage.

**–x** *directory*

tells Perl that the script is embedded in a message. Leading garbage will be discarded until the first line that starts with #! and contains the string "perl". Any meaningful switches on that line will be applied. If a directory name is specified, Perl will switch to that directory before running the script. The **–x** switch controls only the disposal of leading garbage. The script must be terminated with `__END__` if there is trailing garbage to be ignored (the script can process any or all of the trailing garbage via the DATA filehandle if desired).

## ENVIRONMENT

HOME Used if chdir has no argument.

LOGDIR Used if chdir has no argument and HOME is not set.

PATH Used in executing subprocesses, and in finding the script if **–S** is used.

PERL5LIB A colon-separated list of directories in which to look for Perl library files before looking in the standard library and the current directory. If PERL5LIB is not defined, PERLLIB is used. When running taint checks (because the script was running setuid or setgid, or the **–T** switch was used), neither variable is used. The script should instead say

```
use lib "/my/directory";
```

PERL5OPT Command-line options (switches). Switches in this variable are taken as if they were on every Perl command line. Only the **–[DIMUdmw]** switches are allowed. When running taint checks (because the script was running setuid or setgid, or the **–T** switch was used), this variable is ignored.

PERLLIB A colon-separated list of directories in which to look for Perl library files before looking in the standard library and the current directory. If PERL5LIB is defined, PERLLIB is not used.

PERL5DB The command used to load the debugger code. The default is:

```
BEGIN { require 'perl5db.pl' }
```

PERL5SHELL (specific to WIN32 port)
May be set to an alternative shell that perl must use internally for executing "backtick" commands or *system()*. Default is `cmd.exe /x/c` on WindowsNT and `command.com /c` on Windows95. The value is considered to be space delimited. Precede any character that needs to be protected (like a space or backslash) with a backslash.

Note that Perl doesn't use COMSPEC for this purpose because COMSPEC has a high degree of variability among users, leading to portability concerns. Besides, perl can use a shell that may not be fit for interactive use, and setting COMSPEC to such a shell may interfere with the proper functioning of other programs (which usually look in COMSPEC to find a shell fit for interactive use).

PERL_DEBUG_MSTATS
Relevant only if perl is compiled with the malloc included with the perl distribution (that is, if `perl -V:d_mymalloc` is 'define'). If set, this causes memory statistics to be dumped after execution. If set to an integer greater than one, also causes memory statistics to be dumped after compilation.

PERL_DESTRUCT_LEVEL
Relevant only if your perl executable was built with **–DDEBUGGING**, this controls the behavior of global destruction of objects and other references.

Perl also has environment variables that control how Perl handles data specific to particular natural languages. See the *perllocale* manpage.

Apart from these, Perl uses no other environment variables, except to make them available to the script being executed, and to child processes. However, scripts running setuid would do well to execute the following lines before doing anything else, just to keep people honest:

```
$ENV{PATH} = '/bin:/usr/bin';    # or whatever you need
$ENV{SHELL} = '/bin/sh' if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

## NAME

perlfunc – Perl builtin functions

## DESCRIPTION

The functions in this section can serve as terms in an expression. They fall into two major categories: list operators and named unary operators. These differ in their precedence relationship with a following comma. (See the precedence table in the *perlop* manpage.) List operators take more than one argument, while unary operators can never take more than one argument. Thus, a comma terminates the argument of a unary operator, but merely separates the arguments of a list operator. A unary operator generally provides a scalar context to its argument, while a list operator may provide either scalar and list contexts for its arguments. If it does both, the scalar arguments will be first, and the list argument will follow. (Note that there can ever be only one list argument.) For instance, *splice()* has three scalar arguments followed by a list.

In the syntax descriptions that follow, list operators that expect a list (and provide list context for the elements of the list) are shown with LIST as an argument. Such a list may consist of any combination of scalar arguments or list values; the list values will be included in the list as if each individual element were interpolated at that point in the list, forming a longer single-dimensional list value. Elements of the LIST should be separated by commas.

Any function in the list below may be used either with or without parentheses around its arguments. (The syntax descriptions omit the parentheses.) If you use the parentheses, the simple (but occasionally surprising) rule is this: It *LOOKS* like a function, therefore it *IS* a function, and precedence doesn't matter. Otherwise it's a list operator or unary operator, and precedence does matter. And whitespace between the function and left parenthesis doesn't count— so you need to be careful sometimes:

```
print 1+2+4;        # Prints 7.
print(1+2) + 4;     # Prints 3.
print (1+2)+4;      # Also prints 3!
print +(1+2)+4;     # Prints 7.
print ((1+2)+4);    # Prints 7.
```

If you run Perl with the **–w** switch it can warn you about this. For example, the third line above produces:

```
print (...) interpreted as function at - line 1.
Useless use of integer addition in void context at - line 1.
```

For functions that can be used in either a scalar or list context, nonabortive failure is generally indicated in a scalar context by returning the undefined value, and in a list context by returning the null list.

Remember the following important rule: There is **no rule** that relates the behavior of an expression in list context to its behavior in scalar context, or vice versa. It might do two totally different things. Each operator and function decides which sort of value it would be most appropriate to return in a scalar context. Some operators return the length of the list that would have been returned in list context. Some operators return the first value in the list. Some operators return the last value in the list. Some operators return a count of successful operations. In general, they do what you want, unless you want consistency.

An named array in scalar context is quite different from what would at first glance appear to be a list in scalar context. You can't get a list like `(1,2,3)` into being in scalar context, because the compiler knows the context at compile time. It would generate the scalar comma operator there, not the list construction version of the comma. That means it was never a list to start with.

In general, functions in Perl that serve as wrappers for system calls of the same name (like *chown*(2), *fork*(2), *closedir*(2), etc.) all return true when they succeed and `undef` otherwise, as is usually mentioned in the descriptions below. This is different from the C interfaces, which return `-1` on failure. Exceptions to this rule are `wait()`, `waitpid()`, and `syscall()`. System calls also set the special `$!` variable on failure. Other functions do not, except accidentally.

**Perl Functions by Category**

Here are Perl's functions (including things that look like functions, like some keywords and named operators) arranged by category. Some functions appear in more than one place.

Functions for SCALARs or strings
: `chomp`, `chop`, `chr`, `crypt`, `hex`, `index`, `lc`, `lcfirst`, `length`, `oct`, `ord`, `pack`, `q/STRING/`, `qq/STRING/`, `reverse`, `rindex`, `sprintf`, `substr`, `tr///`, `uc`, `ucfirst`, `y///`

Regular expressions and pattern matching
: `m//`, `pos`, `quotemeta`, `s///`, `split`, `study`, `qr//`

Numeric functions
: `abs`, `atan2`, `cos`, `exp`, `hex`, `int`, `log`, `oct`, `rand`, `sin`, `sqrt`, `srand`

Functions for real @ARRAYs
: `pop`, `push`, `shift`, `splice`, `unshift`

Functions for list data
: `grep`, `join`, `map`, `qw/STRING/`, `reverse`, `sort`, `unpack`

Functions for real %HASHes
: `delete`, `each`, `exists`, `keys`, `values`

Input and output functions
: `binmode`, `close`, `closedir`, `dbmclose`, `dbmopen`, `die`, `eof`, `fileno`, `flock`, `format`, `getc`, `print`, `printf`, `read`, `readdir`, `rewinddir`, `seek`, `seekdir`, `select`, `syscall`, `sysread`, `sysseek`, `syswrite`, `tell`, `telldir`, `truncate`, `warn`, `write`

Functions for fixed length data or records
: `pack`, `read`, `syscall`, `sysread`, `syswrite`, `unpack`, `vec`

Functions for filehandles, files, or directories
: `-`*X*, `chdir`, `chmod`, `chown`, `chroot`, `fcntl`, `glob`, `ioctl`, `link`, `lstat`, `mkdir`, `open`, `opendir`, `readlink`, `rename`, `rmdir`, `stat`, `symlink`, `umask`, `unlink`, `utime`

Keywords related to the control flow of your perl program
: `caller`, `continue`, `die`, `do`, `dump`, `eval`, `exit`, `goto`, `last`, `next`, `redo`, `return`, `sub`, `wantarray`

Keywords related to scoping
: `caller`, `import`, `local`, `my`, `package`, `use`

Miscellaneous functions
: `defined`, `dump`, `eval`, `formline`, `local`, `my`, `reset`, `scalar`, `undef`, `wantarray`

Functions for processes and process groups
: `alarm`, `exec`, `fork`, `getpgrp`, `getppid`, `getpriority`, `kill`, `pipe`, `qx/STRING/`, `setpgrp`, `setpriority`, `sleep`, `system`, `times`, `wait`, `waitpid`

Keywords related to perl modules
: `do`, `import`, `no`, `package`, `require`, `use`

Keywords related to classes and object-orientedness
: `bless`, `dbmclose`, `dbmopen`, `package`, `ref`, `tie`, `tied`, `untie`, `use`

Low-level socket functions
: `accept`, `bind`, `connect`, `getpeername`, `getsockname`, `getsockopt`, `listen`, `recv`, `send`, `setsockopt`, `shutdown`, `socket`, `socketpair`

System V interprocess communication functions
: `msgctl`, `msgget`, `msgrcv`, `msgsnd`, `semctl`, `semget`, `semop`, `shmctl`, `shmget`, `shmread`, `shmwrite`

Fetching user and group info
: `endgrent`, `endhostent`, `endnetent`, `endpwent`, `getgrent`, `getgrgid`, `getgrnam`, `getlogin`, `getpwent`, `getpwnam`, `getpwuid`, `setgrent`, `setpwent`

Fetching network info
: `endprotoent`, `endservent`, `gethostbyaddr`, `gethostbyname`, `gethostent`, `getnetbyaddr`, `getnetbyname`, `getnetent`, `getprotobyname`, `getprotobynumber`, `getprotoent`, `getservbyname`, `getservbyport`, `getservent`, `sethostent`, `setnetent`, `setprotoent`, `setservent`

Time-related functions
: `gmtime`, `localtime`, `time`, `times`

Functions new in perl5
: `abs`, `bless`, `chomp`, `chr`, `exists`, `formline`, `glob`, `import`, `lc`, `lcfirst`, `map`, `my`, `no`, `prototype`, `qx`, `qw`, `readline`, `readpipe`, `ref`, `sub*`, `sysopen`, `tie`, `tied`, `uc`, `ucfirst`, `untie`, `use`

  * − `sub` was a keyword in perl4, but in perl5 it is an operator, which can be used in expressions.

Functions obsoleted in perl5
: `dbmclose`, `dbmopen`

**Alphabetical Listing of Perl Functions**

−*X* FILEHANDLE

−*X* EXPR

−*X* A file test, where X is one of the letters listed below. This unary operator takes one argument, either a filename or a filehandle, and tests the associated file to see if something is true about it. If the argument is omitted, tests `$_`, except for `−t`, which tests STDIN. Unless otherwise documented, it returns `1` for TRUE and `''` for FALSE, or the undefined value if the file doesn't exist. Despite the funny names, precedence is the same as any other named unary operator, and the argument may be parenthesized like any other unary operator. The operator may be any of:

```
−r  File is readable by effective uid/gid.
−w  File is writable by effective uid/gid.
−x  File is executable by effective uid/gid.
−o  File is owned by effective uid.

−R  File is readable by real uid/gid.
−W  File is writable by real uid/gid.
−X  File is executable by real uid/gid.
−O  File is owned by real uid.

−e  File exists.
−z  File has zero size.
−s  File has nonzero size (returns size).

−f  File is a plain file.
−d  File is a directory.
−l  File is a symbolic link.
−p  File is a named pipe (FIFO), or Filehandle is a pipe.
−S  File is a socket.
−b  File is a block special file.
−c  File is a character special file.
−t  Filehandle is opened to a tty.
```

```
-u  File has setuid bit set.
-g  File has setgid bit set.
-k  File has sticky bit set.

-T  File is a text file.
-B  File is a binary file (opposite of -T).

-M  Age of file in days when script started.
-A  Same for access time.
-C  Same for inode change time.
```

The interpretation of the file permission operators `-r`, `-R`, `-w`, `-W`, `-x`, and `-X` is based solely on the mode of the file and the uids and gids of the user. There may be other reasons you can't actually read, write, or execute the file, such as AFS access control lists. Also note that, for the superuser, `-r`, `-R`, `-w`, and `-W` always return `1`, and `-x` and `-X` return `1` if any execute bit is set in the mode. Scripts run by the superuser may thus need to do a `stat()` to determine the actual mode of the file, or temporarily set the uid to something else.

Example:

```
while (<>) {
    chop;
    next unless -f $_;      # ignore specials
    #...
}
```

Note that `-s/a/b/` does not do a negated substitution. Saying `-exp($foo)` still works as expected, however— only single letters following a minus are interpreted as file tests.

The `-T` and `-B` switches work as follows. The first block or so of the file is examined for odd characters such as strange control codes or characters with the high bit set. If too many strange characters (>30%) are found, it's a `-B` file, otherwise it's a `-T` file. Also, any file containing null in the first block is considered a binary file. If `-T` or `-B` is used on a filehandle, the current stdio buffer is examined rather than the first block. Both `-T` and `-B` return TRUE on a null file, or a file at EOF when testing a filehandle. Because you have to read a file to do the `-T` test, on most occasions you want to use a `-f` against the file first, as in `next unless -f $file && -T $file`.

If any of the file tests (or either the `stat()` or `lstat()` operators) are given the special filehandle consisting of a solitary underline, then the stat structure of the previous file test (or stat operator) is used, saving a system call. (This doesn't work with `-t`, and you need to remember that *lstat()* and `-l` will leave values in the stat structure for the symbolic link, not the real file.) Example:

```
print "Can do.\n" if -r $a || -w _ || -x _;

stat($filename);
print "Readable\n" if -r _;
print "Writable\n" if -w _;
print "Executable\n" if -x _;
print "Setuid\n" if -u _;
print "Setgid\n" if -g _;
print "Sticky\n" if -k _;
print "Text\n" if -T _;
print "Binary\n" if -B _;
```

abs VALUE

abs Returns the absolute value of its argument. If VALUE is omitted, uses `$_`.

accept NEWSOCKET,GENERICSOCKET

Accepts an incoming socket connect, just as the *accept*(2) system call does. Returns the packed address if it succeeded, FALSE otherwise. See example in the section on *Sockets: Client/Server Communication* in the *perlipc* manpage.

alarm SECONDS

alarm Arranges to have a SIGALRM delivered to this process after the specified number of seconds have elapsed. If SECONDS is not specified, the value stored in `$_` is used. (On some machines, unfortunately, the elapsed time may be up to one second less than you specified because of how seconds are counted.) Only one timer may be counting at once. Each call disables the previous timer, and an argument of `0` may be supplied to cancel the previous timer without starting a new one. The returned value is the amount of time remaining on the previous timer.

For delays of finer granularity than one second, you may use Perl's `syscall()` interface to access *setitimer*(2) if your system supports it, or else see the `select()` entry elsewhere in this document. It is usually a mistake to intermix `alarm()` and `sleep()` calls.

If you want to use `alarm()` to time out a system call you need to use an `eval()`/`die()` pair. You can't rely on the alarm causing the system call to fail with `$!` set to `EINTR` because Perl sets up signal handlers to restart system calls on some systems. Using `eval()`/`die()` always works, modulo the caveats given in the section on *Signals* in the *perlipc* manpage.

```
eval {
    local $SIG{ALRM} = sub { die "alarm\n" }; # NB: \n required
    alarm $timeout;
    $nread = sysread SOCKET, $buffer, $size;
    alarm 0;
};
if ($@) {
    die unless $@ eq "alarm\n";   # propagate unexpected errors
    # timed out
}
else {
    # didn't
}
```

atan2 Y,X

Returns the arctangent of Y/X in the range $-\pi$ to $\pi$.

For the tangent operation, you may use the `POSIX::tan()` function, or use the familiar relation:

```
sub tan { sin($_[0]) / cos($_[0])  }
```

bind SOCKET,NAME

Binds a network address to a socket, just as the bind system call does. Returns TRUE if it succeeded, FALSE otherwise. NAME should be a packed address of the appropriate type for the socket. See the examples in the section on *Sockets: Client/Server Communication* in the *perlipc* manpage.

binmode FILEHANDLE

Arranges for the file to be read or written in "binary" mode in operating systems that distinguish between binary and text files. Files that are not in binary mode have CR LF sequences translated to LF on input and LF translated to CR LF on output. Binmode has no effect under Unix; in

MS−DOS and similarly archaic systems, it may be imperative—otherwise your MS−DOS−damaged C library may mangle your file. The key distinction between systems that need `binmode()` and those that don't is their text file formats. Systems like Unix, MacOS, and Plan9 that delimit lines with a single character, and that encode that character in C as `"\n"`, do not need `binmode()`. The rest need it. If FILEHANDLE is an expression, the value is taken as the name of the filehandle.

bless REF,CLASSNAME

bless REF

This function tells the thingy referenced by REF that it is now an object in the CLASSNAME package—or the current package if no CLASSNAME is specified, which is often the case. It returns the reference for convenience, because a `bless()` is often the last thing in a constructor. Always use the two-argument version if the function doing the blessing might be inherited by a derived class. See the *perltoot* manpage and the *perlobj* manpage for more about the blessing (and blessings) of objects.

caller EXPR

caller

Returns the context of the current subroutine call. In scalar context, returns the caller's package name if there is a caller, that is, if we're in a subroutine or `eval()` or `require()`, and the undefined value otherwise. In list context, returns

```
($package, $filename, $line) = caller;
```

With EXPR, it returns some extra information that the debugger uses to print a stack trace. The value of EXPR indicates how many call frames to go back before the current one.

```
($package, $filename, $line, $subroutine,
 $hasargs, $wantarray, $evaltext, $is_require) = caller($i);
```

Here `$subroutine` may be `"(eval)"` if the frame is not a subroutine call, but an `eval()`. In such a case additional elements `$evaltext` and `$is_require` are set: `$is_require` is true if the frame is created by a `require` or `use` statement, `$evaltext` contains the text of the `eval EXPR` statement. In particular, for a `eval BLOCK` statement, `$filename` is `"(eval)"`, but `$evaltext` is undefined. (Note also that each `use` statement creates a `require` frame inside an `eval EXPR`) frame.

Furthermore, when called from within the DB package, caller returns more detailed information: it sets the list variable `@DB::args` to be the arguments with which the subroutine was invoked.

Be aware that the optimizer might have optimized call frames away before `caller()` had a chance to get the information. That means that `caller(N)` might not return information about the call frame you expect it do, for `N > 1`. In particular, `@DB::args` might have information from the previous time `caller()` was called.

chdir EXPR

Changes the working directory to EXPR, if possible. If EXPR is omitted, changes to home directory. Returns TRUE upon success, FALSE otherwise. See example under `die()`.

chmod LIST

Changes the permissions of a list of files. The first element of the list must be the numerical mode, which should probably be an octal number, and which definitely should *not* a string of octal digits: `0644` is okay, `'0644'` is not. Returns the number of files successfully changed. See also the `oct` entry elsewhere in this documentif all you have is a string.

```
$cnt = chmod 0755, 'foo', 'bar';
chmod 0755, @executables;
$mode = '0644'; chmod $mode, 'foo';      # !!! sets mode to
                                         # --w----r-T
$mode = '0644'; chmod oct($mode), 'foo'; # this is better
$mode = 0644;   chmod $mode, 'foo';      # this is best
```

chomp VARIABLE

chomp LIST

chomp This is a slightly safer version of the `chop` entry elsewhere in this document. It removes any line ending that corresponds to the current value of `$/` (also known as `$INPUT_RECORD_SEPARATOR` in the `English` module). It returns the total number of characters removed from all its arguments. It's often used to remove the newline from the end of an input record when you're worried that the final record may be missing its newline. When in paragraph mode (`$/ = ""`), it removes all trailing newlines from the string. If VARIABLE is omitted, it chomps `$_`. Example:

```
while (<>) {
    chomp;  # avoid \n on last field
    @array = split(/:/);
    # ...
}
```

You can actually chomp anything that's an lvalue, including an assignment:

```
chomp($cwd = `pwd`);
chomp($answer = <STDIN>);
```

If you chomp a list, each element is chomped, and the total number of characters removed is returned.

chop VARIABLE

chop LIST

chop Chops off the last character of a string and returns the character chopped. It's used primarily to remove the newline from the end of an input record, but is much more efficient than `s/\n//` because it neither scans nor copies the string. If VARIABLE is omitted, chops `$_`. Example:

```
while (<>) {
    chop;   # avoid \n on last field
    @array = split(/:/);
    #...
}
```

You can actually chop anything that's an lvalue, including an assignment:

```
chop($cwd = `pwd`);
chop($answer = <STDIN>);
```

If you chop a list, each element is chopped. Only the value of the last `chop()` is returned.

Note that `chop()` returns the last character. To return all but the last character, use `substr($string, 0, -1)`.

chown LIST

Changes the owner (and group) of a list of files. The first two elements of the list must be the *NUMERICAL* uid and gid, in that order. Returns the number of files successfully changed.

```
$cnt = chown $uid, $gid, 'foo', 'bar';
chown $uid, $gid, @filenames;
```

Here's an example that looks up nonnumeric uids in the passwd file:

```
print "User: ";
chop($user = <STDIN>);
print "Files: ";
chop($pattern = <STDIN>);

($login,$pass,$uid,$gid) = getpwnam($user)
    or die "$user not in passwd file";

@ary = glob($pattern);      # expand filenames
chown $uid, $gid, @ary;
```

On most systems, you are not allowed to change the ownership of the file unless you're the superuser, although you should be able to change the group to any of your secondary groups. On insecure systems, these restrictions may be relaxed, but this is not a portable assumption.

chr NUMBER

chr Returns the character represented by that NUMBER in the character set. For example, `chr(65)` is `"A"` in ASCII. For the reverse, use the `ord` entry elsewhere in this document.

If NUMBER is omitted, uses `$_`.

chroot FILENAME

chroot This function works like the system call by the same name: it makes the named directory the new root directory for all further pathnames that begin with a `"/"` by your process and all its children. (It doesn't change your current working directory, which is unaffected.) For security reasons, this call is restricted to the superuser. If FILENAME is omitted, does a `chroot()` to `$_`.

close FILEHANDLE

close Closes the file or pipe associated with the file handle, returning TRUE only if stdio successfully flushes buffers and closes the system file descriptor. Closes the currently selected filehandle if the argument is omitted.

You don't have to close FILEHANDLE if you are immediately going to do another `open()` on it, because `open()` will close it for you. (See `open()`.) However, an explicit `close()` on an input file resets the line counter (`$.`), while the implicit close done by `open()` does not.

If the file handle came from a piped open `close()` will additionally return FALSE if one of the other system calls involved fails or if the program exits with non-zero status. (If the only problem was that the program exited non-zero `$!` will be set to `0`.) Also, closing a pipe waits for the process executing on the pipe to complete, in case you want to look at the output of the pipe afterwards. Closing a pipe explicitly also puts the exit status value of the command into `$?`.

Example:

```
open(OUTPUT, '|sort >foo')  # pipe to sort
    or die "Can't start sort: $!";
#...                        # print stuff to output
close OUTPUT                # wait for sort to finish
    or warn $! ? "Error closing sort pipe: $!"
               : "Exit status $? from sort";
open(INPUT, 'foo')          # get sort's results
    or die "Can't open 'foo' for input: $!";
```

FILEHANDLE may be an expression whose value can be used as an indirect filehandle, usually the real filehandle name.

closedir DIRHANDLE

Closes a directory opened by `opendir()` and returns the success of that system call.

DIRHANDLE may be an expression whose value can be used as an indirect dirhandle, usually the real dirhandle name.

connect SOCKET,NAME

Attempts to connect to a remote socket, just as the connect system call does. Returns TRUE if it succeeded, FALSE otherwise. NAME should be a packed address of the appropriate type for the socket. See the examples in the section on *Sockets: Client/Server Communication* in the *perlipc* manpage.

continue BLOCK

Actually a flow control statement rather than a function. If there is a `continue` BLOCK attached to a BLOCK (typically in a `while` or `foreach`), it is always executed just before the conditional is about to be evaluated again, just like the third part of a `for` loop in C. Thus it can be used to increment a loop variable, even when the loop has been continued via the `next` statement (which is similar to the C `continue` statement).

`last`, `next`, or `redo` may appear within a `continue` block. `last` and `redo` will behave as if they had been executed within the main block. So will `next`, but since it will execute a `continue` block, it may be more entertaining.

```
while (EXPR) {
    ### redo always comes here
    do_something;
} continue {
    ### next always comes here
    do_something_else;
    # then back the top to re-check EXPR
}
### last always comes here
```

Omitting the `continue` section is semantically equivalent to using an empty one, logically enough. In that case, `next` goes directly back to check the condition at the top of the loop.

cos EXPR

Returns the cosine of EXPR (expressed in radians). If EXPR is omitted, takes cosine of `$_`.

For the inverse cosine operation, you may use the `POSIX::acos()` function, or use this relation:

```
sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }
```

crypt PLAINTEXT,SALT

Encrypts a string exactly like the *crypt*(3) function in the C library (assuming that you actually have a version there that has not been extirpated as a potential munition). This can prove useful for checking the password file for lousy passwords, amongst other things. Only the guys wearing white hats should do this.

Note that `crypt()` is intended to be a one-way function, much like breaking eggs to make an omelette. There is no (known) corresponding decrypt function. As a result, this function isn't all that useful for cryptography. (For that, see your nearby CPAN mirror.)

Here's an example that makes sure that whoever runs this program knows their own password:

```
$pwd = (getpwuid($<))[1];
$salt = substr($pwd, 0, 2);

system "stty -echo";
print "Password: ";
chop($word = <STDIN>);
print "\n";
system "stty echo";

if (crypt($word, $salt) ne $pwd) {
    die "Sorry...\n";
} else {
    print "ok\n";
}
```

Of course, typing in your own password to whoever asks you for it is unwise.

dbmclose HASH

[This function has been superseded by the `untie()` function.]

Breaks the binding between a DBM file and a hash.

dbmopen HASH,DBNAME,MODE

[This function has been superseded by the `tie()` function.]

This binds a *dbm*(3), *ndbm*(3), *sdbm*(3), *gdbm*(3), or Berkeley DB file to a hash. HASH is the name of the hash. (Unlike normal `open()`, the first argument is *NOT* a filehandle, even though it looks like one). DBNAME is the name of the database (without the *.dir* or *.pag* extension if any). If the database does not exist, it is created with protection specified by MODE (as modified by the `umask()`). If your system supports only the older DBM functions, you may perform only one `dbmopen()` in your program. In older versions of Perl, if your system had neither DBM nor ndbm, calling `dbmopen()` produced a fatal error; it now falls back to *sdbm*(3).

If you don't have write access to the DBM file, you can only read hash variables, not set them. If you want to test whether you can write, either use file tests or try setting a dummy hash entry inside an `eval()`, which will trap the error.

Note that functions such as `keys()` and `values()` may return huge lists when used on large DBM files. You may prefer to use the `each()` function to iterate over large DBM files. Example:

```
# print out history file offsets
dbmopen(%HIST,'/usr/lib/news/history',0666);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
dbmclose(%HIST);
```

See also the *AnyDBM_File* manpage for a more general description of the pros and cons of the various dbm approaches, as well as the *DB_File* manpage for a particularly rich implementation.

defined EXPR

defined Returns a Boolean value telling whether EXPR has a value other than the undefined value `undef`. If EXPR is not present, `$_` will be checked.

Many operations return `undef` to indicate failure, end of file, system error, uninitialized variable, and other exceptional conditions. This function allows you to distinguish `undef` from other values. (A simple Boolean test will not distinguish among `undef`, zero, the empty string, and `"0"`, which are all equally false.) Note that since `undef` is a valid scalar, its presence doesn't *necessarily* indicate an exceptional condition: `pop()` returns `undef` when its argument is an empty array, *or* when the element to return happens to be `undef`.

You may also use `defined()` to check whether a subroutine exists, by saying `defined &func` without parentheses. On the other hand, use of `defined()` upon aggregates (hashes and arrays) is not guaranteed to produce intuitive results, and should probably be avoided.

When used on a hash element, it tells you whether the value is defined, not whether the key exists in the hash. Use the `exists` entry elsewhere in this documentfor the latter purpose.

Examples:

```
print if defined $switch{'D'};
print "$val\n" while defined($val = pop(@ary));
die "Can't readlink $sym: $!"
    unless defined($value = readlink $sym);
sub foo { defined &$bar ? &$bar(@_) : die "No bar"; }
$debugging = 0 unless defined $debugging;
```

Note: Many folks tend to overuse `defined()`, and then are surprised to discover that the number `0` and `""` (the zero-length string) are, in fact, defined values. For example, if you say

```
"ab" =~ /a(.*)b/;
```

The pattern match succeeds, and `$1` is defined, despite the fact that it matched "nothing". But it didn't really match nothing— rather, it matched something that happened to be `0` characters long. This is all very above-board and honest. When a function returns an undefined value, it's an admission that it couldn't give you an honest answer. So you should use `defined()` only when you're questioning the integrity of what you're trying to do. At other times, a simple comparison to `0` or `""` is what you want.

Currently, using `defined()` on an entire array or hash reports whether memory for that aggregate has ever been allocated. So an array you set to the empty list appears undefined initially, and one that once was full and that you then set to the empty list still appears defined. You should instead use a simple test for size:

```
if (@an_array) { print "has array elements\n" }
if (%a_hash)   { print "has hash members\n"   }
```

Using `undef()` on these, however, does clear their memory and then report them as not defined anymore, but you shouldn't do that unless you don't plan to use them again, because it saves time when you load them up again to have memory already ready to be filled. The normal way to free up space used by an aggregate is to assign the empty list.

This counterintuitive behavior of `defined()` on aggregates may be changed, fixed, or broken in a future release of Perl.

See also the `undef`, `exists`, and `ref` entries elsewhere in this document.

delete EXPR

Deletes the specified *key*(s) and their associated values from a hash. For each key, returns the deleted value associated with that key, or the undefined value if there was no such key. Deleting from `$ENV{}` modifies the environment. Deleting from a hash tied to a DBM file deletes the entry from the DBM file. (But deleting from a `tie()`d hash doesn't necessarily return anything.)

The following deletes all the values of a hash:

```
foreach $key (keys %HASH) {
    delete $HASH{$key};
}
```

And so does this:

```
delete @HASH{keys %HASH}
```

(But both of these are slower than just assigning the empty list, or using `undef()`.) Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash element lookup or hash slice:

```
delete $ref->[$x][$y]{$key};
delete @{$ref->[$x][$y]}{$key1, $key2, @morekeys};
```

die LIST Outside an `eval()`, prints the value of LIST to `STDERR` and exits with the current value of `$!` (errno). If `$!` is `0`, exits with the value of `($? >> 8)` (backtick ‘command‘ status). If `($? >> 8)` is `0`, exits with `255`. Inside an `eval(),` the error message is stuffed into `$@` and the `eval()` is terminated with the undefined value. This makes `die()` the way to raise an exception.

Equivalent examples:

```
die "Can't cd to spool: $!\n" unless chdir '/usr/spool/news';
chdir '/usr/spool/news' or die "Can't cd to spool: $!\n"
```

If the value of EXPR does not end in a newline, the current script line number and input line number (if any) are also printed, and a newline is supplied. Hint: sometimes appending `", stopped"` to your message will cause it to make better sense when the string `"at foo line 123"` is appended. Suppose you are running script “canasta”.

```
die "/etc/games is no good";
die "/etc/games is no good, stopped";
```

produce, respectively

```
/etc/games is no good at canasta line 123.
/etc/games is no good, stopped at canasta line 123.
```

See also `exit()` and `warn()`.

If LIST is empty and `$@` already contains a value (typically from a previous eval) that value is reused after appending `"\t...propagated"`. This is useful for propagating exceptions:

```
eval { ... };
die unless $@ =~ /Expected exception/;
```

If `$@` is empty then the string `"Died"` is used.

You can arrange for a callback to be run just before the `die()` does its deed, by setting the `$SIG{__DIE__}` hook. The associated handler will be called with the error text and can change the error message, if it sees fit, by calling `die()` again. See the section on *$SIG{expr}* in the *perlvar* manpage for details on setting `%SIG` entries, and the section on *eval BLOCK* for some examples.

Note that the `$SIG{__DIE__}` hook is called even inside *eval()*ed blocks/strings. If one wants the hook to do nothing in such situations, put

```
die @_ if $^S;
```

as the first line of the handler (see the section on *$^S* in the *perlvar* manpage).

do BLOCK

Not really a function. Returns the value of the last command in the sequence of commands indicated by BLOCK. When modified by a loop modifier, executes the BLOCK once before testing the loop condition. (On other statements the loop modifiers test the conditional first.)

do SUBROUTINE(LIST)

A deprecated form of subroutine call. See the *perlsub* manpage.

do EXPR

Uses the value of EXPR as a filename and executes the contents of the file as a Perl script. Its primary use is to include subroutines from a Perl subroutine library.

```
do 'stat.pl';
```

is just like

```
scalar eval `cat stat.pl`;
```

except that it's more efficient and concise, keeps track of the current filename for error messages, and searches all the **–I** libraries if the file isn't in the current directory (see also the `@INC` array in the section on *Predefined Names* in the *perlvar* manpage). It is also different in how code evaluated with `do FILENAME` doesn't see lexicals in the enclosing scope like `eval STRING` does. It's the same, however, in that it does reparse the file every time you call it, so you probably don't want to do this inside a loop.

If `do` cannot read the file, it returns undef and sets `$!` to the error. If `do` can read the file but cannot compile it, it returns undef and sets an error message in `$@`. If the file is successfully compiled, `do` returns the value of the last expression evaluated.

Note that inclusion of library modules is better done with the `use()` and `require()` operators, which also do automatic error checking and raise an exception if there's a problem.

You might like to use `do` to read in a program configuration file. Manual error checking can be done this way:

```
# read in config files: system first, then user
for $file ("/share/prog/defaults.rc",
           "$ENV{HOME}/.someprogrc") {
    unless ($return = do $file) {
        warn "couldn't parse $file: $@" if $@;
        warn "couldn't do $file: $!"    unless defined $return;
        warn "couldn't run $file"       unless $return;
    }
}
```

dump LABEL

This causes an immediate core dump. Primarily this is so that you can use the **undump** program to turn your core dump into an executable binary after having initialized all your variables at the beginning of the program. When the new binary is executed it will begin by executing a `goto LABEL` (with all the restrictions that `goto` suffers). Think of it as a goto with an intervening core dump and reincarnation. If `LABEL` is omitted, restarts the program from the top. WARNING: Any files opened at the time of the dump will NOT be open any more when the program is reincarnated, with possible resulting confusion on the part of Perl. See also **–u** option in the *perlrun* manpage.

Example:

```
#!/usr/bin/perl
require 'getopt.pl';
require 'stat.pl';
%days = (
    'Sun' => 1,
    'Mon' => 2,
    'Tue' => 3,
    'Wed' => 4,
    'Thu' => 5,
    'Fri' => 6,
    'Sat' => 7,
);

dump QUICKSTART if $ARGV[0] eq '-d';

QUICKSTART:
Getopt('f');
```

This operator is largely obsolete, partly because it's very hard to convert a core file into an executable, and because the real perl-to-C compiler has superseded it.

each HASH

When called in list context, returns a 2-element list consisting of the key and value for the next element of a hash, so that you can iterate over it. When called in scalar context, returns the key for only the "next" element in the hash. (Note: Keys may be `"0"` or `""`, which are logically false; you may wish to avoid constructs like `while ($k = each %foo) {}` for this reason.)

Entries are returned in an apparently random order. When the hash is entirely read, a null array is returned in list context (which when assigned produces a FALSE (`0`) value), and `undef` in scalar context. The next call to `each()` after that will start iterating again. There is a single iterator for each hash, shared by all `each()`, `keys()`, and `values()` function calls in the program; it can be reset by reading all the elements from the hash, or by evaluating `keys HASH` or `values HASH`. If you add or delete elements of a hash while you're iterating over it, you may get entries skipped or duplicated, so don't.

The following prints out your environment like the *printenv*(1) program, only in a different order:

```
while (($key,$value) = each %ENV) {
    print "$key=$value\n";
}
```

See also `keys()` and `values()`.

eof FILEHANDLE

eof ()

eof

Returns 1 if the next read on FILEHANDLE will return end of file, or if FILEHANDLE is not open. FILEHANDLE may be an expression whose value gives the real filehandle. (Note that this function actually reads a character and then `ungetc()`s it, so isn't very useful in an interactive context.) Do not read from a terminal file (or call `eof(FILEHANDLE)` on it) after end-of-file is reached. Filetypes such as terminals may lose the end-of-file condition if you do.

An `eof` without an argument uses the last file read as argument. Using `eof()` with empty parentheses is very different. It indicates the pseudo file formed of the files listed on the command line, i.e., `eof()` is reasonable to use inside a `while (<>)` loop to detect the end of only the last file. Use `eof(ARGV)` or eof without the parentheses to test *EACH* file in a while (<>) loop. Examples:

```
# reset line numbering on each input file
while (<>) {
    next if /^\s*#/;        # skip comments
    print "$.\t$_";
} continue {
    close ARGV  if eof;     # Not eof()!
}

# insert dashes just before last line of last file
while (<>) {
    if (eof()) {            # check for end of current file
        print "--------------\n";
        close(ARGV);        # close or break; is needed if we
                            # are reading from the terminal
    }
    print;
}
```

Practical hint: you almost never need to use `eof` in Perl, because the input operators return false values when they run out of data, or if there was an error.

eval EXPR

eval BLOCK

In the first form, the return value of EXPR is parsed and executed as if it were a little Perl program. The value of the expression (which is itself determined within scalar context) is first parsed, and if there weren't any errors, executed in the context of the current Perl program, so that any variable settings or subroutine and format definitions remain afterwards. Note that the value is parsed every time the eval executes. If EXPR is omitted, evaluates `$_`. This form is typically used to delay parsing and subsequent execution of the text of EXPR until run time.

In the second form, the code within the BLOCK is parsed only once— at the same time the code surrounding the eval itself was parsed— and executed within the context of the current Perl program. This form is typically used to trap exceptions more efficiently than the first (see below), while also providing the benefit of checking the code within BLOCK at compile time.

The final semicolon, if any, may be omitted from the value of EXPR or within the BLOCK.

In both forms, the value returned is the value of the last expression evaluated inside the mini-program; a return statement may be also used, just as with subroutines. The expression providing the return value is evaluated in void, scalar, or list context, depending on the context of the eval itself. See the `wantarray` entry elsewhere in this documentfor more on how the evaluation context can be determined.

If there is a syntax error or runtime error, or a `die()` statement is executed, an undefined value is returned by `eval()`, and `$@` is set to the error message. If there was no error, `$@` is guaranteed to be a null string. Beware that using `eval()` neither silences perl from printing warnings to STDERR, nor does it stuff the text of warning messages into `$@`. To do either of those, you have to use the `$SIG{__WARN__}` facility. See the `warn` entry elsewhere in this document and the *perlvar* manpage.

Note that, because `eval()` traps otherwise-fatal errors, it is useful for determining whether a particular feature (such as `socket()` or `symlink()`) is implemented. It is also Perl's exception trapping mechanism, where the die operator is used to raise exceptions.

If the code to be executed doesn't vary, you may use the eval-BLOCK form to trap run-time errors without incurring the penalty of recompiling each time. The error, if any, is still returned in `$@`. Examples:

```
# make divide-by-zero nonfatal
eval { $answer = $a / $b; }; warn $@ if $@;

# same thing, but less efficient
eval '$answer = $a / $b'; warn $@ if $@;

# a compile-time error
eval { $answer = };                 # WRONG

# a run-time error
eval '$answer =';   # sets $@
```

When using the `eval{}` form as an exception trap in libraries, you may wish not to trigger any `__DIE__` hooks that user code may have installed. You can use the `local $SIG{__DIE__}` construct for this purpose, as shown in this example:

```
# a very private exception trap for divide-by-zero
eval { local $SIG{'__DIE__'}; $answer = $a / $b; };
warn $@ if $@;
```

This is especially significant, given that `__DIE__` hooks can call `die()` again, which has the effect of changing their error messages:

```
# __DIE__ hooks may modify error messages
{
   local $SIG{'__DIE__'} =
          sub { (my $x = $_[0]) =~ s/foo/bar/g; die $x };
   eval { die "foo lives here" };
   print $@ if $@;                # prints "bar lives here"
}
```

With an `eval()`, you should be especially careful to remember what's being looked at when:

```
eval $x;            # CASE 1
eval "$x";          # CASE 2

eval '$x';          # CASE 3
eval { $x };        # CASE 4

eval "\$$x++";      # CASE 5
$$x++;              # CASE 6
```

Cases 1 and 2 above behave identically: they run the code contained in the variable `$x`. (Although case 2 has misleading double quotes making the reader wonder what else might be happening (nothing is).) Cases 3 and 4 likewise behave in the same way: they run the code `'$x'`, which does nothing but return the value of `$x`. (Case 4 is preferred for purely visual reasons, but it also has the advantage of compiling at compile-time instead of at run-time.) Case 5 is a place where normally you *WOULD* like to use double quotes, except that in this particular situation, you can just use symbolic references instead, as in case 6.

exec LIST

exec PROGRAM LIST

The `exec()` function executes a system command *AND NEVER RETURNS* – use `system()` instead of `exec()` if you want it to return. It fails and returns FALSE only if the command does not exist *and* it is executed directly instead of via your system's command shell (see below).

Since it's a common mistake to use `exec()` instead of `system()`, Perl warns you if there is a following statement which isn't `die()`, `warn()`, or `exit()` (if `-w` is set – but you always do that). If you *really* want to follow an `exec()` with some other statement, you can use one of these styles to avoid the warning:

```
exec ('foo')   or print STDERR "couldn't exec foo: $!";
{ exec ('foo') }; print STDERR "couldn't exec foo: $!";
```

If there is more than one argument in LIST, or if LIST is an array with more than one value, calls *execvp*(3) with the arguments in LIST. If there is only one scalar argument or an array with one element in it, the argument is checked for shell metacharacters, and if there are any, the entire argument is passed to the system's command shell for parsing (this is `/bin/sh -c` on Unix platforms, but varies on other platforms). If there are no shell metacharacters in the argument, it is split into words and passed directly to `execvp()`, which is more efficient. Note: `exec()` and `system()` do not flush your output buffer, so you may need to set `$|` to avoid lost output. Examples:

```
exec '/bin/echo', 'Your arguments are: ', @ARGV;
exec "sort $outfile | uniq";
```

If you don't really want to execute the first argument, but want to lie to the program you are executing about its own name, you can specify the program you actually want to run as an "indirect object" (without a comma) in front of the LIST. (This always forces interpretation of the LIST as a multivalued list, even if there is only a single scalar in the list.) Example:

```
$shell = '/bin/csh';
exec $shell '-sh';          # pretend it's a login shell
```

or, more directly,

```
exec {'/bin/csh'} '-sh';    # pretend it's a login shell
```

When the arguments get executed via the system shell, results will be subject to its quirks and capabilities. See the section on *'STRING'* in the *perlop* manpage for details.

Using an indirect object with `exec()` or `system()` is also more secure. This usage forces interpretation of the arguments as a multivalued list, even if the list had just one argument. That way you're safe from the shell expanding wildcards or splitting up words with whitespace in them.

```
@args = ( "echo surprise" );

system @args;               # subject to shell escapes
                            # if @args == 1
system { $args[0] } @args;  # safe even with one-arg list
```

The first version, the one without the indirect object, ran the *echo* program, passing it `"surprise"` an argument. The second version didn't—it tried to run a program literally called *"echo surprise"*, didn't find it, and set `$?` to a non-zero value indicating failure.

Note that `exec()` will not call your `END` blocks, nor will it call any `DESTROY` methods in your objects.

exists EXPR

Returns TRUE if the specified hash key exists in its hash array, even if the corresponding value is undefined.

```
print "Exists\n" if exists $array{$key};
print "Defined\n" if defined $array{$key};
print "True\n" if $array{$key};
```

A hash element can be TRUE only if it's defined, and defined if it exists, but the reverse doesn't necessarily hold true.

Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash key lookup:

```
if (exists $ref->{"A"}{"B"}{$key}) { ... }
```

Although the last element will not spring into existence just because its existence was tested, intervening ones will. Thus `$ref->{"A"}` `$ref->{"B"}` will spring into existence due to the existence test for a `$key` element. This autovivification may be fixed in a later release.

exit EXPR

Evaluates EXPR and exits immediately with that value. (Actually, it calls any defined `END` routines first, but the `END` routines may not abort the exit. Likewise any object destructors that need to be called are called before exit.) Example:

```
$ans = <STDIN>;
exit 0 if $ans =~ /^[Xx]/;
```

See also `die()`. If EXPR is omitted, exits with `0` status. The only universally portable values for EXPR are `0` for success and `1` for error; all other values are subject to unpredictable interpretation depending on the environment in which the Perl program is running.

You shouldn't use `exit()` to abort a subroutine if there's any chance that someone might want to trap whatever error happened. Use `die()` instead, which can be trapped by an `eval()`.

All `END{}` blocks are run at exit time. See the *perlsub* manpage for details.

exp EXPR

exp Returns *e* (the natural logarithm base) to the power of EXPR. If EXPR is omitted, gives `exp($_)`.

fcntl FILEHANDLE,FUNCTION,SCALAR

Implements the *fcntl*(2) function. You'll probably have to say

```
use Fcntl;
```

first to get the correct constant definitions. Argument processing and value return works just like `ioctl()` below. For example:

```
use Fcntl;
fcntl($filehandle, F_GETFL, $packed_return_buffer)
    or die "can't fcntl F_GETFL: $!";
```

You don't have to check for `defined()` on the return from `fnctl()`. Like `ioctl()`, it maps a `0` return from the system call into "`0` but true" in Perl. This string is true in boolean context and `0` in numeric context. It is also exempt from the normal **−w** warnings on improper numeric conversions.

Note that `fcntl()` will produce a fatal error if used on a machine that doesn't implement *fcntl*(2).

fileno FILEHANDLE

Returns the file descriptor for a filehandle. This is useful for constructing bitmaps for `select()` and low-level POSIX tty-handling operations. If FILEHANDLE is an expression, the value is taken as an indirect filehandle, generally its name.

You can use this to find out whether two handles refer to the same underlying descriptor:

```
if (fileno(THIS) == fileno(THAT)) {
    print "THIS and THAT are dups\n";
}
```

flock FILEHANDLE,OPERATION

Calls *flock*(2), or an emulation of it, on FILEHANDLE. Returns TRUE for success, FALSE on failure. Produces a fatal error if used on a machine that doesn't implement *flock*(2), *fcntl*(2) locking, or *lockf*(3). `flock()` is Perl's portable file locking interface, although it locks only entire files, not records.

On many platforms (including most versions or clones of Unix), locks established by `flock()` are **merely advisory**. Such discretionary locks are more flexible, but offer fewer guarantees. This means that files locked with `flock()` may be modified by programs that do not also use `flock()`. Windows NT and OS/2 are among the platforms which enforce mandatory locking. See your local documentation for details.

OPERATION is one of LOCK_SH, LOCK_EX, or LOCK_UN, possibly combined with LOCK_NB. These constants are traditionally valued 1, 2, 8 and 4, but you can use the symbolic names if import them from the Fcntl module, either individually, or as a group using the ':flock' tag. LOCK_SH requests a shared lock, LOCK_EX requests an exclusive lock, and LOCK_UN releases a previously requested lock. If LOCK_NB is added to LOCK_SH or LOCK_EX then `flock()` will return immediately rather than blocking waiting for the lock (check the return status to see if you got it).

To avoid the possibility of mis-coordination, Perl flushes FILEHANDLE before (un)locking it.

Note that the emulation built with *lockf*(3) doesn't provide shared locks, and it requires that FILEHANDLE be open with write intent. These are the semantics that *lockf*(3) implements. Most (all?) systems implement *lockf*(3) in terms of *fcntl*(2) locking, though, so the differing semantics shouldn't bite too many people.

Note also that some versions of `flock()` cannot lock things over the network; you would need to use the more system-specific `fcntl()` for that. If you like you can force Perl to ignore your system's *flock*(2) function, and so provide its own *fcntl*(2)–based emulation, by passing the switch `-Ud_flock` to the *Configure* program when you configure perl.

Here's a mailbox appender for BSD systems.

```
use Fcntl ':flock'; # import LOCK_* constants

sub lock {
    flock(MBOX,LOCK_EX);
    # and, in case someone appended
    # while we were waiting...
    seek(MBOX, 0, 2);
}

sub unlock {
    flock(MBOX,LOCK_UN);
}

open(MBOX, ">>/usr/spool/mail/$ENV{'USER'}")
        or die "Can't open mailbox: $!";
```

```
lock();
print MBOX $msg,"\n\n";
unlock();
```

See also the *DB_File* manpage for other *flock()* examples.

fork
: Does a *fork* (2) system call. Returns the child pid to the parent process, `0` to the child process, or `undef` if the fork is unsuccessful.

Note: unflushed buffers remain unflushed in both processes, which means you may need to set `$|` ($AUTOFLUSH in English) or call the `autoflush()` method of `IO::Handle` to avoid duplicate output.

If you `fork()` without ever waiting on your children, you will accumulate zombies:

```
$SIG{CHLD} = sub { wait };
```

There's also the double-fork trick (error checking on `fork()` returns omitted);

```
unless ($pid = fork) {
    unless (fork) {
        exec "what you really wanna do";
        die "no exec";
        # ... or ...
        ## (some_perl_code_here)
        exit 0;
    }
    exit 0;
}
waitpid($pid,0);
```

See also the *perlipc* manpage for more examples of forking and reaping moribund children.

Note that if your forked child inherits system file descriptors like STDIN and STDOUT that are actually connected by a pipe or socket, even if you exit, then the remote server (such as, say, httpd or rsh) won't think you're done. You should reopen those to */dev/null* if it's any issue.

format
: Declare a picture format for use by the `write()` function. For example:

```
format Something =
    Test: @<<<<<<<< @||||| @>>>>>
          $str,     $%,    '$' . int($num)
.

$str = "widget";
$num = $cost/$quantity;
$~ = 'Something';
write;
```

See the *perlform* manpage for many details and examples.

formline PICTURE,LIST
: This is an internal function used by `formats`, though you may call it, too. It formats (see the *perlform* manpage) a list of values according to the contents of PICTURE, placing the output into the format output accumulator, `$^A` (or `$ACCUMULATOR` in English). Eventually, when a `write()` is done, the contents of `$^A` are written to some filehandle, but you could also read `$^A` yourself and then set `$^A` back to `""`. Note that a format typically does one `formline()` per line of form, but the `formline()` function itself doesn't care how many newlines are embedded in the PICTURE. This means that the `~` and `~~` tokens will treat the entire PICTURE as

a single line. You may therefore need to use multiple formlines to implement a single record format, just like the format compiler.

Be careful if you put double quotes around the picture, because an "`@`" character may be taken to mean the beginning of an array name. `formline()` always returns TRUE. See the *perlform* manpage for other examples.

getc FILEHANDLE

getc Returns the next character from the input file attached to FILEHANDLE, or the undefined value at end of file, or if there was an error. If FILEHANDLE is omitted, reads from STDIN. This is not particularly efficient. It cannot be used to get unbuffered single-characters, however. For that, try something more like:

```
if ($BSD_STYLE) {
    system "stty cbreak </dev/tty >/dev/tty 2>&1";
}
else {
    system "stty", '-icanon', 'eol', "\001";
}

$key = getc(STDIN);

if ($BSD_STYLE) {
    system "stty -cbreak </dev/tty >/dev/tty 2>&1";
}
else {
    system "stty", 'icanon', 'eol', '^@'; # ASCII null
}
print "\n";
```

Determination of whether `$BSD_STYLE` should be set is left as an exercise to the reader.

The `POSIX::getattr()` function can do this more portably on systems purporting POSIX compliance. See also the `Term::ReadKey` module from your nearest CPAN site; details on CPAN can be found on the `CPAN` entry in the *perlmod* manpage.

getlogin Implements the C library function of the same name, which on most systems returns the current login from */etc/utmp*, if any. If null, use `getpwuid()`.

```
$login = getlogin || getpwuid($<) || "Kilroy";
```

Do not consider `getlogin()` for authentication: it is not as secure as `getpwuid()`.

getpeername SOCKET

Returns the packed sockaddr address of other end of the SOCKET connection.

```
use Socket;
$hersockaddr    = getpeername(SOCK);
($port, $iaddr) = unpack_sockaddr_in($hersockaddr);
$herhostname    = gethostbyaddr($iaddr, AF_INET);
$herstraddr     = inet_ntoa($iaddr);
```

getpgrp PID

Returns the current process group for the specified PID. Use a PID of `0` to get the current process group for the current process. Will raise an exception if used on a machine that doesn't implement *getpgrp*(2). If PID is omitted, returns process group of current process. Note that the POSIX version of `getpgrp()` does not accept a PID argument, so only `PID==0` is truly portable.

getppid Returns the process id of the parent process.

getpriority WHICH,WHO
: Returns the current priority for a process, a process group, or a user. (See the *getpriority(2)* manpage.) Will raise a fatal exception if used on a machine that doesn't implement *getpriority* (2).

getpwnam NAME

getgrnam NAME

gethostbyname NAME

getnetbyname NAME

getprotobyname NAME

getpwuid UID

getgrgid GID

getservbyname NAME,PROTO

gethostbyaddr ADDR,ADDRTYPE

getnetbyaddr ADDR,ADDRTYPE

getprotobynumber NUMBER

getservbyport PORT,PROTO

getpwent

getgrent

gethostent

getnetent

getprotoent

getservent

setpwent

setgrent

sethostent STAYOPEN

setnetent STAYOPEN

setprotoent STAYOPEN

setservent STAYOPEN

endpwent

endgrent

endhostent

endnetent

endprotoent

endservent
: These routines perform the same functions as their counterparts in the system library. In list context, the return values from the various get routines are as follows:

```
($name,$passwd,$uid,$gid,
   $quota,$comment,$gcos,$dir,$shell,$expire) = getpw*
($name,$passwd,$gid,$members) = getgr*
($name,$aliases,$addrtype,$length,@addrs) = gethost*
($name,$aliases,$addrtype,$net) = getnet*
($name,$aliases,$proto) = getproto*
($name,$aliases,$port,$proto) = getserv*
```

(If the entry doesn't exist you get a null list.)

In scalar context, you get the name, unless the function was a lookup by name, in which case you get the other thing, whatever it is. (If the entry doesn't exist you get the undefined value.) For example:

```
$uid   = getpwnam($name);
$name  = getpwuid($num);
$name  = getpwent();
$gid   = getgrnam($name);
$name  = getgrgid($num;
$name  = getgrent();
#etc.
```

In *getpw*()* the fields `$quota`, `$comment`, and `$expire` are special cases in the sense that in many systems they are unsupported. If the `$quota` is unsupported, it is an empty scalar. If it is supported, it usually encodes the disk quota. If the `$comment` field is unsupported, it is an empty scalar. If it is supported it usually encodes some administrative comment about the user. In some systems the `$quota` field may be `$change` or `$age`, fields that have to do with password aging. In some systems the `$comment` field may be `$class`. The `$expire` field, if present, encodes the expiration period of the account or the password. For the availability and the exact meaning of these fields in your system, please consult your *getpwnam*(3) documentation and your *pwd.h* file. You can also find out from within Perl which meaning your `$quota` and `$comment` fields have and whether you have the `$expire` field by using the `Config` module and the values `d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment`, and `d_pwexpire`.

The `$members` value returned by *getgr*()* is a space separated list of the login names of the members of the group.

For the *gethost*()* functions, if the `h_errno` variable is supported in C, it will be returned to you via `$?` if the function call fails. The `@addrs` value returned by a successful call is a list of the raw addresses returned by the corresponding system library call. In the Internet domain, each address is four bytes long and you can unpack it by saying something like:

```
($a,$b,$c,$d) = unpack('C4',$addr[0]);
```

If you get tired of remembering which element of the return list contains which return value, by-name interfaces are also provided in modules: `File::stat`, `Net::hostent`, `Net::netent`, `Net::protoent`, `Net::servent`, `Time::gmtime`, `Time::localtime`, and `User::grent`. These override the normal built-in, replacing them with versions that return objects with the appropriate names for each field. For example:

```
use File::stat;
use User::pwent;
$is_his = (stat($filename)->uid == pwent($whoever)->uid);
```

Even though it looks like they're the same method calls (uid), they aren't, because a `File::stat` object is different from a `User::pwent` object.

getsockname SOCKET

Returns the packed sockaddr address of this end of the SOCKET connection.

```
use Socket;
$mysockaddr = getsockname(SOCK);
($port, $myaddr) = unpack_sockaddr_in($mysockaddr);
```

getsockopt SOCKET,LEVEL,OPTNAME

Returns the socket option requested, or undef if there is an error.

glob EXPR

glob Returns the value of EXPR with filename expansions such as the standard Unix shell */bin/sh* would do. This is the internal function implementing the `<*.c>` operator, but you can use it directly. If EXPR is omitted, `$_` is used. The `<*.c>` operator is discussed in more detail in the section on *I/O Operators* in the *perlop* manpage.

gmtime EXPR

Converts a time as returned by the time function to a 9-element array with the time localized for the standard Greenwich time zone. Typically used as follows:

```
#  0    1    2     3     4    5     6     7     8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
                                        gmtime(time);
```

All array elements are numeric, and come straight out of a struct tm. In particular this means that `$mon` has the range `0..11` and `$wday` has the range `0..6` with sunday as day `0`. Also, `$year` is the number of years since 1900, that is, `$year` is `123` in year 2023, *not* simply the last two digits of the year.

If EXPR is omitted, does `gmtime(time())`.

In scalar context, returns the *ctime*(3) value:

```
$now_string = gmtime;  # e.g., "Thu Oct 13 04:54:34 1994"
```

Also see the `timegm()` function provided by the `Time::Local` module, and the *strftime*(3) function available via the POSIX module.

This scalar value is **not** locale dependent, see the *perllocale* manpage, but instead a Perl builtin. Also see the `Time::Local` module, and the *strftime*(3) and *mktime*(3) function available via the POSIX module. To get somewhat similar but locale dependent date strings, set up your locale environment variables appropriately (please see the *perllocale* manpage) and try for example:

```
use POSIX qw(strftime);
    $now_string = strftime "%a %b %e %H:%M:%S %Y", gmtime;
```

Note that the `%a` and `%b`, the short forms of the day of the week and the month of the year, may not necessarily be three characters wide.

goto LABEL

goto EXPR

goto &NAME

The `goto-LABEL` form finds the statement labeled with LABEL and resumes execution there. It may not be used to go into any construct that requires initialization, such as a subroutine or a `foreach` loop. It also can't be used to go into a construct that is optimized away, or to get out of a block or subroutine given to `sort()`. It can be used to go almost anywhere else within the dynamic scope, including out of subroutines, but it's usually better to use some other construct such as `last` or `die()`. The author of Perl has never felt the need to use this form of `goto` (in

Perl, that is— C is another matter).

The `goto-EXPR` form expects a label name, whose scope will be resolved dynamically. This allows for computed `gotos` per FORTRAN, but isn't necessarily recommended if you're optimizing for maintainability:

```
goto ("FOO", "BAR", "GLARCH")[$i];
```

The `goto-&NAME` form is highly magical, and substitutes a call to the named subroutine for the currently running subroutine. This is used by `AUTOLOAD` subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place (except that any modifications to `@_` in the current subroutine are propagated to the other subroutine.) After the `goto`, not even `caller()` will be able to tell that this routine was called first.

grep BLOCK LIST

grep EXPR,LIST

This is similar in spirit to, but not the same as, *grep*(1) and its relatives. In particular, it is not limited to using regular expressions.

Evaluates the BLOCK or EXPR for each element of LIST (locally setting `$_` to each element) and returns the list value consisting of those elements for which the expression evaluated to TRUE. In a scalar context, returns the number of times the expression was TRUE.

```
@foo = grep(!/^#/, @bar);    # weed out comments
```

or equivalently,

```
@foo = grep {!/^#/} @bar;    # weed out comments
```

Note that, because `$_` is a reference into the list value, it can be used to modify the elements of the array. While this is useful and supported, it can cause bizarre results if the LIST is not a named array. Similarly, grep returns aliases into the original list, much like the way that a for loop's index variable aliases the list elements. That is, modifying an element of a list returned by grep (for example, in a `foreach`, `map()` or another `grep()`) actually modifies the element in the original list.

See also the `map` entry elsewhere in this documentfor an array composed of the results of the BLOCK or EXPR.

hex EXPR

hex Interprets EXPR as a hex string and returns the corresponding value. (To convert strings that might start with either 0 or 0x see the `oct` entry elsewhere in this document.) If EXPR is omitted, uses `$_`.

```
print hex '0xAf'; # prints '175'
print hex 'aF';   # same
```

import There is no builtin `import()` function. It is just an ordinary method (subroutine) defined (or inherited) by modules that wish to export names to another module. The `use()` function calls the `import()` method for the package used. See also the `use()` entry elsewhere in this documentthe *perlmod* manpage, and the *Exporter* manpage.

index STR,SUBSTR,POSITION

index STR,SUBSTR

Returns the position of the first occurrence of SUBSTR in STR at or after POSITION. If POSITION is omitted, starts searching from the beginning of the string. The return value is based at `0` (or whatever you've set the `$[` variable to— but don't do that). If the substring is not found, returns one less than the base, ordinarily `-1`.

int EXPR

int
Returns the integer portion of EXPR. If EXPR is omitted, uses `$_`. You should not use this for rounding, because it truncates towards `0`, and because machine representations of floating point numbers can sometimes produce counterintuitive results. Usually `sprintf()` or `printf()`, or the `POSIX::floor` or `POSIX::ceil` functions, would serve you better.

ioctl FILEHANDLE,FUNCTION,SCALAR
Implements the *ioctl*(2) function. You'll probably have to say

```
require "ioctl.ph"; # probably in /usr/local/lib/perl/ioctl.ph
```

first to get the correct function definitions. If *ioctl.ph* doesn't exist or doesn't have the correct definitions you'll have to roll your own, based on your C header files such as *<sys/ioctl.h>*. (There is a Perl script called **h2ph** that comes with the Perl kit that may help you in this, but it's nontrivial.) SCALAR will be read and/or written depending on the FUNCTION— a pointer to the string value of SCALAR will be passed as the third argument of the actual `ioctl()` call. (If SCALAR has no string value but does have a numeric value, that value will be passed rather than a pointer to the string value. To guarantee this to be TRUE, add a `0` to the scalar before using it.) The `pack()` and `unpack()` functions are useful for manipulating the values of structures used by `ioctl()`. The following example sets the erase character to DEL.

```
require 'ioctl.ph';
$getp = &TIOCGETP;
die "NO TIOCGETP" if $@ || !$getp;
$sgttyb_t = "ccccs";                # 4 chars and a short
if (ioctl(STDIN,$getp,$sgttyb)) {
    @ary = unpack($sgttyb_t,$sgttyb);
    $ary[2] = 127;
    $sgttyb = pack($sgttyb_t,@ary);
    ioctl(STDIN,&TIOCSETP,$sgttyb)
        || die "Can't ioctl: $!";
}
```

The return value of `ioctl()` (and `fcntl()`) is as follows:

```
        if OS returns:          then Perl returns:
            -1                    undefined value
             0                  string "0 but true"
        anything else               that number
```

Thus Perl returns TRUE on success and FALSE on failure, yet you can still easily determine the actual value returned by the operating system:

```
($retval = ioctl(...)) || ($retval = -1);
printf "System returned %d\n", $retval;
```

The special string "`0` but true" is excempt from **–w** complaints about improper numeric conversions.

join EXPR,LIST
Joins the separate strings of LIST into a single string with fields separated by the value of EXPR, and returns the string. Example:

```
$_ = join(':', $login,$passwd,$uid,$gid,$gcos,$home,$shell);
```

See the `split` entry elsewhere in this document.

keys HASH

Returns a list consisting of all the keys of the named hash. (In a scalar context, returns the number of keys.) The keys are returned in an apparently random order, but it is the same order as either the `values()` or `each()` function produces (given that the hash has not been modified). As a side effect, it resets HASH's iterator.

Here is yet another way to print your environment:

```
@keys = keys %ENV;
@values = values %ENV;
while ($#keys >= 0) {
    print pop(@keys), '=', pop(@values), "\n";
}
```

or how about sorted by key:

```
foreach $key (sort(keys %ENV)) {
    print $key, '=', $ENV{$key}, "\n";
}
```

To sort an array by value, you'll need to use a `sort()` function. Here's a descending numeric sort of a hash by its values:

```
foreach $key (sort { $hash{$b} <=> $hash{$a} } keys %hash) {
    printf "%4d %s\n", $hash{$key}, $key;
}
```

As an lvalue `keys()` allows you to increase the number of hash buckets allocated for the given hash. This can gain you a measure of efficiency if you know the hash is going to get big. (This is similar to pre-extending an array by assigning a larger number to $#array.) If you say

```
keys %hash = 200;
```

then `%hash` will have at least 200 buckets allocated for it—256 of them, in fact, since it rounds up to the next power of two. These buckets will be retained even if you do `%hash = ()`, use `undef %hash` if you want to free the storage while `%hash` is still in scope. You can't shrink the number of buckets allocated for the hash using `keys()` in this way (but you needn't worry about doing this by accident, as trying has no effect).

kill LIST

Sends a signal to a list of processes. The first element of the list must be the signal to send. Returns the number of processes successfully signaled.

```
$cnt = kill 1, $child1, $child2;
kill 9, @goners;
```

Unlike in the shell, in Perl if the *SIGNAL* is negative, it kills process groups instead of processes. (On System V, a negative *PROCESS* number will also kill process groups, but that's not portable.) That means you usually want to use positive not negative signals. You may also use a signal name in quotes. See the section on *Signals* in the *perlipc* manpage for details.

last LABEL

last

The `last` command is like the `break` statement in C (as used in loops); it immediately exits the loop in question. If the LABEL is omitted, the command refers to the innermost enclosing loop. The `continue` block, if any, is not executed:

```
LINE: while (<STDIN>) {
    last LINE if /^$/;      # exit when done with header
    #...
}
```

See also the `continue` entry elsewhere in this documentfor an illustration of how `last`, `next`, and `redo` work.

lc EXPR

lc Returns an lowercased version of EXPR. This is the internal function implementing the `\L` escape in double-quoted strings. Respects current `LC_CTYPE` locale if `use locale` in force. See the *perllocale* manpage.

If EXPR is omitted, uses `$_`.

lcfirst EXPR

lcfirst Returns the value of EXPR with the first character lowercased. This is the internal function implementing the `\l` escape in double-quoted strings. Respects current `LC_CTYPE` locale if `use locale` in force. See the *perllocale* manpage.

If EXPR is omitted, uses `$_`.

length EXPR

length Returns the length in bytes of the value of EXPR. If EXPR is omitted, returns length of `$_`.

link OLDFILE,NEWFILE

Creates a new filename linked to the old filename. Returns TRUE for success, FALSE otherwise.

listen SOCKET,QUEUESIZE

Does the same thing that the listen system call does. Returns TRUE if it succeeded, FALSE otherwise. See example in the section on *Sockets: Client/Server Communication* in the *perlipc* manpage.

local EXPR

A local modifies the listed variables to be local to the enclosing block, file, or eval. If more than one value is listed, the list must be placed in parentheses. See the section on *Temporary Values via local()* in the *perlsub* manpage for details, including issues with tied arrays and hashes.

You really probably want to be using `my()` instead, because `local()` isn't what most people think of as "local". See the section on *Private Variables via my()* in the *perlsub* manpage for details.

localtime EXPR

Converts a time as returned by the time function to a 9-element array with the time analyzed for the local time zone. Typically used as follows:

```
#  0    1    2     3     4    5     6     7     8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
                                            localtime(time);
```

All array elements are numeric, and come straight out of a struct tm. In particular this means that `$mon` has the range `0..11` and `$wday` has the range `0..6` with sunday as day `0`. Also, `$year` is the number of years since 1900, that is, `$year` is `123` in year 2023, and *not* simply the last two digits of the year.

If EXPR is omitted, uses the current time (`localtime(time)`).

In scalar context, returns the *ctime* (3) value:

```
$now_string = localtime;  # e.g., "Thu Oct 13 04:54:34 1994"
```

This scalar value is **not** locale dependent, see the *perllocale* manpage, but instead a Perl builtin. Also see the `Time::Local` module, and the *strftime* (3) and *mktime* (3) function available via the POSIX module. To get somewhat similar but locale dependent date strings, set up your locale environment variables appropriately (please see the *perllocale* manpage) and try for example:

```
use POSIX qw(strftime);
    $now_string = strftime "%a %b %e %H:%M:%S %Y", localtime;
```

Note that the `%a` and `%b`, the short forms of the day of the week and the month of the year, may not necessarily be three characters wide.

log EXPR

log Returns the natural logarithm (base *e*) of EXPR. If EXPR is omitted, returns log of `$_`.

lstat FILEHANDLE

lstat EXPR

lstat Does the same thing as the `stat()` function (including setting the special _ filehandle) but stats a symbolic link instead of the file the symbolic link points to. If symbolic links are unimplemented on your system, a normal `stat()` is done.

If EXPR is omitted, stats `$_`.

m// The match operator. See the *perlop* manpage.

map BLOCK LIST

map EXPR,LIST

Evaluates the BLOCK or EXPR for each element of LIST (locally setting `$_` to each element) and returns the list value composed of the results of each such evaluation. Evaluates BLOCK or EXPR in a list context, so each element of LIST may produce zero, one, or more elements in the returned value.

```
@chars = map(chr, @nums);
```

translates a list of numbers to the corresponding characters. And

```
%hash = map { getkey($_) => $_ } @array;
```

is just a funny way to write

```
%hash = ();
foreach $_ (@array) {
    $hash{getkey($_)} = $_;
}
```

Note that, because `$_` is a reference into the list value, it can be used to modify the elements of the array. While this is useful and supported, it can cause bizarre results if the LIST is not a named array. See also the `grep` entry elsewhere in this documentfor an array composed of those items of the original list for which the BLOCK or EXPR evaluates to true.

mkdir FILENAME,MODE

Creates the directory specified by FILENAME, with permissions specified by MODE (as modified by umask). If it succeeds it returns TRUE, otherwise it returns FALSE and sets `$!` (errno).

msgctl ID,CMD,ARG

Calls the System V IPC function *msgctl* (2). You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If CMD is `IPC_STAT`, then ARG must be a variable which will hold the returned `msqid_ds` structure. Returns like `ioctl()`: the undefined value for error, "`0` but true" for zero, or the actual return value otherwise. See also `IPC::SysV` and `IPC::Semaphore::Msg` documentation.

msgget KEY,FLAGS
: Calls the System V IPC function *msgget*(2). Returns the message queue id, or the undefined value if there is an error. See also `IPC::SysV` and `IPC::SysV::Msg` documentation.

msgsnd ID,MSG,FLAGS
: Calls the System V IPC function msgsnd to send the message MSG to the message queue ID. MSG must begin with the long integer message type, which may be created with `pack("l", $type)`. Returns TRUE if successful, or FALSE if there is an error. See also `IPC::SysV` and `IPC::SysV::Msg` documentation.

msgrcv ID,VAR,SIZE,TYPE,FLAGS
: Calls the System V IPC function msgrcv to receive a message from message queue ID into variable VAR with a maximum message size of SIZE. Note that if a message is received, the message type will be the first thing in VAR, and the maximum length of VAR is SIZE plus the size of the message type. Returns TRUE if successful, or FALSE if there is an error. See also `IPC::SysV` and `IPC::SysV::Msg` documentation.

my EXPR
: A `my()` declares the listed variables to be local (lexically) to the enclosing block, file, or `eval()`. If more than one value is listed, the list must be placed in parentheses. See the section on *Private Variables via my()* in the *perlsub* manpage for details.

next LABEL

next
: The `next` command is like the `continue` statement in C; it starts the next iteration of the loop:

```
LINE: while (<STDIN>) {
    next LINE if /^#/;       # discard comments
    #...
}
```

Note that if there were a `continue` block on the above, it would get executed even on discarded lines. If the LABEL is omitted, the command refers to the innermost enclosing loop.

See also the `continue` entry elsewhere in this documentfor an illustration of how `last`, `next`, and `redo` work.

no Module LIST
: See the the `use` entry elsewhere in this documentfunction, which `no` is the opposite of.

oct EXPR

oct
: Interprets EXPR as an octal string and returns the corresponding value. (If EXPR happens to start off with `0x`, interprets it as a hex string instead.) The following will handle decimal, octal, and hex in the standard Perl or C notation:

```
$val = oct($val) if $val =~ /^0/;
```

If EXPR is omitted, uses `$_`. This function is commonly used when a string such as `644` needs to be converted into a file mode, for example. (Although perl will automatically convert strings into numbers as needed, this automatic conversion assumes base 10.)

open FILEHANDLE,EXPR

open FILEHANDLE
: Opens the file whose filename is given by EXPR, and associates it with FILEHANDLE. If FILEHANDLE is an expression, its value is used as the name of the real filehandle wanted. If EXPR is omitted, the scalar variable of the same name as the FILEHANDLE contains the filename. (Note that lexical variables— those declared with `my()`--will not work for this purpose; so if you're

using `my()`, specify EXPR in your call to open.)

If the filename begins with `'<'` or nothing, the file is opened for input. If the filename begins with `'>'`, the file is truncated and opened for output, being created if necessary. If the filename begins with `'>>'`, the file is opened for appending, again being created if necessary. You can put a `'+'` in front of the `'>'` or `'<'` to indicate that you want both read and write access to the file; thus `'+<'` is almost always preferred for read/write updates— the `'+>'` mode would clobber the file first. You can't usually use either read-write mode for updating textfiles, since they have variable length records. See the **–i** switch in the *perlrun* manpage for a better approach.

The prefix and the filename may be separated with spaces. These various prefixes correspond to the *fopen* (3) modes of `'r'`, `'r+'`, `'w'`, `'w+'`, `'a'`, and `'a+'`.

If the filename begins with `'|'`, the filename is interpreted as a command to which output is to be piped, and if the filename ends with a `'|'`, the filename is interpreted See the section on *Using open() for IPC* in the *perlipc* manpage for more examples of this. (You are not allowed to `open()` to a command that pipes both in *and* out, but see the *IPC::Open2* manpage, the *IPC::Open3* manpage, and the section on *Bidirectional Communication* in the *perlipc* manpage for alternatives.)

Opening `'-'` opens STDIN and opening `'>-'` opens STDOUT. Open returns nonzero upon success, the undefined value otherwise. If the `open()` involved a pipe, the return value happens to be the pid of the subprocess.

If you're unfortunate enough to be running Perl on a system that distinguishes between text files and binary files (modern operating systems don't care), then you should check out the `binmode` entry elsewhere in this documentfor tips for dealing with this. The key distinction between systems that need `binmode()` and those that don't is their text file formats. Systems like Unix, MacOS, and Plan9, which delimit lines with a single character, and which encode that character in C as `"\n"`, do not need `binmode()`. The rest need it.

When opening a file, it's usually a bad idea to continue normal execution if the request failed, so `open()` is frequently used in connection with `die()`. Even if `die()` won't do what you want (say, in a CGI script, where you want to make a nicely formatted error message (but there are modules that can help with that problem)) you should always check the return value from opening a file. The infrequent exception is when working with an unopened filehandle is actually what you want to do.

Examples:

```
$ARTICLE = 100;
open ARTICLE or die "Can't find article $ARTICLE: $!\n";
while (<ARTICLE>) {...

open(LOG, '>>/usr/spool/news/twitlog'); # (log is reserved)
# if the open fails, output is discarded

open(DBASE, '+<dbase.mine')             # open for update
    or die "Can't open 'dbase.mine' for update: $!";

open(ARTICLE, "caesar <$article |")     # decrypt article
    or die "Can't start caesar: $!";

open(EXTRACT, "|sort >/tmp/Tmp$$")      # $$ is our process id
    or die "Can't start sort: $!";

# process argument list of files along with any includes
```

```
foreach $file (@ARGV) {
    process($file, 'fh00');
}

sub process {
    my($filename, $input) = @_;
    $input++;               # this is a string increment
    unless (open($input, $filename)) {
        print STDERR "Can't open $filename: $!\n";
        return;
    }

    local $_;
    while (<$input>) {              # note use of indirection
        if (/^#include "(.*)"/) {
            process($1, $input);
            next;
        }
        #...                # whatever
    }
}
```

You may also, in the Bourne shell tradition, specify an EXPR beginning with `'>&'`, in which case the rest of the string is interpreted as the name of a filehandle (or file descriptor, if numeric) to be duped and opened. You may use `&` after >, >>, <, +>, +>>, and +<. The mode you specify should match the mode of the original filehandle. (Duping a filehandle does not take into account any existing contents of stdio buffers.) Here is a script that saves, redirects, and restores STDOUT and STDERR:

```
#!/usr/bin/perl
open(OLDOUT, ">&STDOUT");
open(OLDERR, ">&STDERR");

open(STDOUT, ">foo.out") || die "Can't redirect stdout";
open(STDERR, ">&STDOUT") || die "Can't dup stdout";

select(STDERR); $| = 1;     # make unbuffered
select(STDOUT); $| = 1;     # make unbuffered

print STDOUT "stdout 1\n";  # this works for
print STDERR "stderr 1\n";  # subprocesses too

close(STDOUT);
close(STDERR);

open(STDOUT, ">&OLDOUT");
open(STDERR, ">&OLDERR");

print STDOUT "stdout 2\n";
print STDERR "stderr 2\n";
```

If you specify `'<&=N'`, where `N` is a number, then Perl will do an equivalent of C's `fdopen()` of that file descriptor; this is more parsimonious of file descriptors. For example:

```
open(FILEHANDLE, "<&=$fd")
```

If you open a pipe on the command `'-'`, i.e., either `'|-'` or `'-|'`, then there is an implicit fork done, and the return value of open is the pid of the child within the parent process, and `0` within the child process. (Use `defined($pid)` to determine whether the open was successful.) The filehandle behaves normally for the parent, but i/o to that filehandle is piped from/to the STDOUT/STDIN of the child process. In the child process the filehandle isn't opened— i/o happens from/to the new STDOUT or STDIN. Typically this is used like the normal piped open when you want to exercise more control over just how the pipe command gets executed, such as when you are running setuid, and don't want to have to scan shell commands for metacharacters. The following pairs are more or less equivalent:

```
open(FOO, "|tr '[a-z]' '[A-Z]'");
open(FOO, "|-") || exec 'tr', '[a-z]', '[A-Z]';

open(FOO, "cat -n '$file'|");
open(FOO, "-|") || exec 'cat', '-n', $file;
```

See the section on *Safe Pipe Opens* in the *perlipc* manpage for more examples of this.

NOTE: On any operation that may do a fork, any unflushed buffers remain unflushed in both processes, which means you may need to set `$|` to avoid duplicate output.

Closing any piped filehandle causes the parent process to wait for the child to finish, and returns the status value in `$?`.

The filename passed to open will have leading and trailing whitespace deleted, and the normal redirection characters honored. This property, known as "magic open", can often be used to good effect. A user could specify a filename of *"rsh cat file |"*, or you could change certain filenames as needed:

```
$filename =~ s/(.*\.gz)\s*$/gzip -dc < $1|/;
open(FH, $filename) or die "Can't open $filename: $!";
```

However, to open a file with arbitrary weird characters in it, it's necessary to protect any leading and trailing whitespace:

```
$file =~ s#^(\s)#./$1#;
open(FOO, "< $file\0");
```

If you want a "real" C `open()` (see the *open(2)* manpage on your system), then you should use the `sysopen()` function, which involves no such magic. This is another way to protect your filenames from interpretation. For example:

```
use IO::Handle;
sysopen(HANDLE, $path, O_RDWR|O_CREAT|O_EXCL)
    or die "sysopen $path: $!";
$oldfh = select(HANDLE); $| = 1; select($oldfh);
print HANDLE "stuff $$\n");
seek(HANDLE, 0, 0);
print "File contains: ", <HANDLE>;
```

Using the constructor from the `IO::Handle` package (or one of its subclasses, such as `IO::File` or `IO::Socket`), you can generate anonymous filehandles that have the scope of whatever variables hold references to them, and automatically close whenever and however you leave that scope:

```
use IO::File;
#...
sub read_myfile_munged {
    my $ALL = shift;
    my $handle = new IO::File;
    open($handle, "myfile") or die "myfile: $!";
    $first = <$handle>
        or return ();     # Automatically closed here.
    mung $first or die "mung failed";       # Or here.
    return $first, <$handle> if $ALL;       # Or here.
    $first;                                 # Or here.
}
```

See the `seek()` entry elsewhere in this documentfor some details about mixing reading and writing.

opendir DIRHANDLE,EXPR
: Opens a directory named EXPR for processing by `readdir()`, `telldir()`, `seekdir()`, `rewinddir()`, and `closedir()`. Returns TRUE if successful. DIRHANDLEs have their own namespace separate from FILEHANDLEs.

ord EXPR

ord
: Returns the numeric ascii value of the first character of EXPR. If EXPR is omitted, uses `$_`. For the reverse, see the `chr` entry elsewhere in this document.

pack TEMPLATE,LIST
: Takes an array or list of values and packs it into a binary structure, returning the string containing the structure. The TEMPLATE is a sequence of characters that give the order and type of values, as follows:

```
A   An ascii string, will be space padded.
a   An ascii string, will be null padded.
b   A bit string (ascending bit order, like vec()).
B   A bit string (descending bit order).
h   A hex string (low nybble first).
H   A hex string (high nybble first).

c   A signed char value.
C   An unsigned char value.

s   A signed short value.
S   An unsigned short value.
      (This 'short' is _exactly_ 16 bits, which may differ from
       what a local C compiler calls 'short'.)

i   A signed integer value.
I   An unsigned integer value.
      (This 'integer' is _at_least_ 32 bits wide.  Its exact
       size depends on what a local C compiler calls 'int',
       and may even be larger than the 'long' described in
       the next item.)
```

```
l   A signed long value.
L   An unsigned long value.
      (This 'long' is _exactly_ 32 bits, which may differ from
       what a local C compiler calls 'long'.)

n   A short in "network" (big-endian) order.
N   A long in "network" (big-endian) order.
v   A short in "VAX" (little-endian) order.
V   A long in "VAX" (little-endian) order.
      (These 'shorts' and 'longs' are _exactly_ 16 bits and
       _exactly_ 32 bits, respectively.)

f   A single-precision float in the native format.
d   A double-precision float in the native format.

p   A pointer to a null-terminated string.
P   A pointer to a structure (fixed-length string).

u   A uuencoded string.

w   A BER compressed integer.  Its bytes represent an unsigned
    integer in base 128, most significant digit first, with as
    few digits as possible.  Bit eight (the high bit) is set
    on each byte except the last.

x   A null byte.
X   Back up a byte.
@   Null fill to absolute position.
```

Each letter may optionally be followed by a number giving a repeat count. With all types except `"a"`, `"A"`, `"b"`, `"B"`, `"h"`, `"H"`, and `"P"` the pack function will gobble up that many values from the LIST. A `*` for the repeat count means to use however many items are left. The `"a"` and `"A"` types gobble just one value, but pack it as a string of length count, padding with nulls or spaces as necessary. (When unpacking, `"A"` strips trailing spaces and nulls, but `"a"` does not.) Likewise, the `"b"` and `"B"` fields pack a string that many bits long. The `"h"` and `"H"` fields pack a string that many nybbles long. The `"p"` type packs a pointer to a null- terminated string. You are responsible for ensuring the string is not a temporary value (which can potentially get deallocated before you get around to using the packed result). The `"P"` packs a pointer to a structure of the size indicated by the length. A NULL pointer is created if the corresponding value for `"p"` or `"P"` is `undef`. Real numbers (floats and doubles) are in the native machine format only; due to the multiplicity of floating formats around, and the lack of a standard "network" representation, no facility for interchange has been made. This means that packed floating point data written on one machine may not be readable on another − even if both use IEEE floating point arithmetic (as the endian-ness of the memory representation is not part of the IEEE spec). Note that Perl uses doubles internally for all numeric calculation, and converting from double into float and thence back to double again will lose precision (i.e., `unpack("f", pack("f", $foo))` will not in general equal `$foo`).

Examples:

```
$foo = pack("cccc",65,66,67,68);
# foo eq "ABCD"
$foo = pack("c4",65,66,67,68);
# same thing
```

```
$foo = pack("ccxxcc",65,66,67,68);
# foo eq "AB\0\0CD"

$foo = pack("s2",1,2);
# "\1\0\2\0" on little-endian
# "\0\1\0\2" on big-endian

$foo = pack("a4","abcd","x","y","z");
# "abcd"

$foo = pack("aaaa","abcd","x","y","z");
# "axyz"

$foo = pack("a14","abcdefg");
# "abcdefg\0\0\0\0\0\0\0"

$foo = pack("i9pl", gmtime);
# a real struct tm (on my system anyway)

sub bintodec {
    unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}
```

The same template may generally also be used in the unpack function.

package

package NAMESPACE

Declares the compilation unit as being in the given namespace. The scope of the package declaration is from the declaration itself through the end of the enclosing block (the same scope as the `local()` operator). All further unqualified dynamic identifiers will be in this namespace. A package statement affects only dynamic variables— including those you've used `local()` on— but *not* lexical variables created with `my()`. Typically it would be the first declaration in a file to be included by the `require` or `use` operator. You can switch into a package in more than one place; it merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and filehandles in other packages by prefixing the identifier with the package name and a double colon: `$Package::Variable`. If the package name is null, the `main` package as assumed. That is, `$::sail` is equivalent to `$main::sail`.

If NAMESPACE is omitted, then there is no current package, and all identifiers must be fully qualified or lexicals. This is stricter than `use strict`, since it also extends to function names.

See the section on *Packages* in the *perlmod* manpage for more information about packages, modules, and classes. See the *perlsub* manpage for other scoping issues.

pipe READHANDLE,WRITEHANDLE

Opens a pair of connected pipes like the corresponding system call. Note that if you set up a loop of piped processes, deadlock can occur unless you are very careful. In addition, note that Perl's pipes use stdio buffering, so you may need to set `$|` to flush your WRITEHANDLE after each command, depending on the application.

See the *IPC::Open2* manpage, the *IPC::Open3* manpage, and the section on *Bidirectional Communication* in the *perlipc* manpage for examples of such things.

pop ARRAY

pop
: Pops and returns the last value of the array, shortening the array by 1. Has a similar effect to

```
$tmp = $ARRAY[$#ARRAY--];
```

If there are no elements in the array, returns the undefined value. If ARRAY is omitted, pops the `@ARGV` array in the main program, and the `@_` array in subroutines, just like `shift()`.

pos SCALAR

pos
: Returns the offset of where the last `m//g` search left off for the variable is in question (`$_` is used when the variable is not specified). May be modified to change that offset. Such modification will also influence the `\G` zero-width assertion in regular expressions. See the *perlre* manpage and the *perlop* manpage.

print FILEHANDLE LIST

print LIST

print
: Prints a string or a comma-separated list of strings. Returns TRUE if successful. FILEHANDLE may be a scalar variable name, in which case the variable contains the name of or a reference to the filehandle, thus introducing one level of indirection. (NOTE: If FILEHANDLE is a variable and the next token is a term, it may be misinterpreted as an operator unless you interpose a + or put parentheses around the arguments.) If FILEHANDLE is omitted, prints by default to standard output (or to the last selected output channel—see the `select` entry elsewhere in this document). If LIST is also omitted, prints `$_` to the currently selected output channel. To set the default output channel to something other than STDOUT use the select operation. Note that, because print takes a LIST, anything in the LIST is evaluated in list context, and any subroutine that you call will have one or more of its expressions evaluated in list context. Also be careful not to follow the print keyword with a left parenthesis unless you want the corresponding right parenthesis to terminate the arguments to the print—interpose a + or put parentheses around all the arguments.

Note that if you're storing FILEHANDLES in an array or other expression, you will have to use a block returning its value instead:

```
print { $files[$i] } "stuff\n";
print { $OK ? STDOUT : STDERR } "stuff\n";
```

printf FILEHANDLE FORMAT, LIST

printf FORMAT, LIST
: Equivalent to `print FILEHANDLE sprintf(FORMAT, LIST)`, except that `$\` (the output record separator) is not appended. The first argument of the list will be interpreted as the `printf()` format. If `use locale` is in effect, the character used for the decimal point in formatted real numbers is affected by the LC_NUMERIC locale. See the *perllocale* manpage.

Don't fall into the trap of using a `printf()` when a simple `print()` would do. The `print()` is more efficient and less error prone.

prototype FUNCTION
: Returns the prototype of a function as a string (or `undef` if the function has no prototype). FUNCTION is a reference to, or the name of, the function whose prototype you want to retrieve.

If FUNCTION is a string starting with `CORE::`, the rest is taken as a name for Perl builtin. If builtin is not *overridable* (such as `qw//`) or its arguments cannot be expressed by a prototype (such as `system()`) – in other words, the builtin does not behave like a Perl function – returns `undef`. Otherwise, the string describing the equivalent prototype is returned.

push ARRAY,LIST
: Treats ARRAY as a stack, and pushes the values of LIST onto the end of ARRAY. The length of ARRAY increases by the length of LIST. Has the same effect as

```
for $value (LIST) {
    $ARRAY[++$#ARRAY] = $value;
}
```

but is more efficient. Returns the new number of elements in the array.

q/STRING/

qq/STRING/

qr/STRING/

qx/STRING/

qw/STRING/
Generalized quotes. See the *perlop* manpage.

quotemeta EXPR

quotemeta
Returns the value of EXPR with all non-alphanumeric characters backslashed. (That is, all characters not matching `/[A-Za-z_0-9]/` will be preceded by a backslash in the returned string, regardless of any locale settings.) This is the internal function implementing the `\Q` escape in double-quoted strings.

If EXPR is omitted, uses `$_`.

rand EXPR

rand
Returns a random fractional number greater than or equal to `0` and less than the value of EXPR. (EXPR should be positive.) If EXPR is omitted, the value `1` is used. Automatically calls `srand()` unless `srand()` has already been called. See also `srand()`.

(Note: If your rand function consistently returns numbers that are too large or too small, then your version of Perl was probably compiled with the wrong number of RANDBITS.)

read FILEHANDLE,SCALAR,LENGTH,OFFSET

read FILEHANDLE,SCALAR,LENGTH
Attempts to read LENGTH bytes of data into variable SCALAR from the specified FILEHANDLE. Returns the number of bytes actually read, `0` at end of file, or undef if there was an error. SCALAR will be grown or shrunk to the length actually read. An OFFSET may be specified to place the read data at some other place than the beginning of the string. This call is actually implemented in terms of stdio's *fread*(3) call. To get a true *read*(2) system call, see `sysread()`.

readdir DIRHANDLE
Returns the next directory entry for a directory opened by `opendir()`. If used in list context, returns all the rest of the entries in the directory. If there are no more entries, returns an undefined value in scalar context or a null list in list context.

If you're planning to filetest the return values out of a `readdir()`, you'd better prepend the directory in question. Otherwise, because we didn't `chdir()` there, it would have been testing the wrong file.

```
opendir(DIR, $some_dir) || die "can't opendir $some_dir: $!";
@dots = grep { /^\./ && -f "$some_dir/$_" } readdir(DIR);
closedir DIR;
```

readline EXPR
Reads from the filehandle whose typeglob is contained in EXPR. In scalar context, a single line is read and returned. In list context, reads until end-of-file is reached and returns a list of lines (however you've defined lines with `$/` or `$INPUT_RECORD_SEPARATOR`). This is the internal

function implementing the <EXPR> operator, but you can use it directly. The <EXPR> operator is discussed in more detail in the section on *I/O Operators* in the *perlop* manpage.

```
$line = <STDIN>;
$line = readline(*STDIN);          # same thing
```

readlink EXPR

readlink Returns the value of a symbolic link, if symbolic links are implemented. If not, gives a fatal error. If there is some system error, returns the undefined value and sets `$!` (errno). If EXPR is omitted, uses `$_`.

readpipe EXPR

EXPR is executed as a system command. The collected standard output of the command is returned. In scalar context, it comes back as a single (potentially multi-line) string. In list context, returns a list of lines (however you've defined lines with `$/` or `$INPUT_RECORD_SEPARATOR`). This is the internal function implementing the `qx/EXPR/` operator, but you can use it directly. The `qx/EXPR/` operator is discussed in more detail in the section on *I/O Operators* in the *perlop* manpage.

recv SOCKET,SCALAR,LEN,FLAGS

Receives a message on a socket. Attempts to receive LENGTH bytes of data into variable SCALAR from the specified SOCKET filehandle. Actually does a C `recvfrom()`, so that it can return the address of the sender. Returns the undefined value if there's an error. SCALAR will be grown or shrunk to the length actually read. Takes the same flags as the system call of the same name. See the section on *UDP: Message Passing* in the *perlipc* manpage for examples.

redo LABEL

redo The `redo` command restarts the loop block without evaluating the conditional again. The `continue` block, if any, is not executed. If the LABEL is omitted, the command refers to the innermost enclosing loop. This command is normally used by programs that want to lie to themselves about what was just input:

```
# a simpleminded Pascal comment stripper
# (warning: assumes no { or } in strings)
LINE: while (<STDIN>) {
    while (s|({.*}.*){.*}|$1 |) {}
    s|{.*}| |;
    if (s|{.*| |) {
        $front = $_;
        while (<STDIN>) {
            if (/}/) {      # end of comment?
                s|^|$front\{|;
                redo LINE;
            }
        }
    }
    print;
}
```

See also the `continue` entry elsewhere in this documentfor an illustration of how `last`, `next`, and `redo` work.

ref EXPR

ref Returns a TRUE value if EXPR is a reference, FALSE otherwise. If EXPR is not specified, `$_` will be used. The value returned depends on the type of thing the reference is a reference to. Builtin types include:

```
REF
SCALAR
ARRAY
HASH
CODE
GLOB
```

If the referenced object has been blessed into a package, then that package name is returned instead. You can think of `ref()` as a `typeof()` operator.

```
if (ref($r) eq "HASH") {
    print "r is a reference to a hash.\n";
}
if (!ref($r)) {
    print "r is not a reference at all.\n";
}
```

See also the *perlref* manpage.

rename OLDNAME,NEWNAME
: Changes the name of a file. Returns `1` for success, `0` otherwise. Will not work across file system boundaries.

require EXPR

require
: Demands some semantics specified by EXPR, or by `$_` if EXPR is not supplied. If EXPR is numeric, demands that the current version of Perl (`$]` or `$PERL_VERSION`) be equal or greater than EXPR.

Otherwise, demands that a library file be included if it hasn't already been included. The file is included via the do-FILE mechanism, which is essentially just a variety of `eval()`. Has semantics similar to the following subroutine:

```
sub require {
    my($filename) = @_;
    return 1 if $INC{$filename};
    my($realfilename,$result);
    ITER: {
        foreach $prefix (@INC) {
            $realfilename = "$prefix/$filename";
            if (-f $realfilename) {
                $result = do $realfilename;
                last ITER;
            }
        }
        die "Can't find $filename in \@INC";
    }
    die $@ if $@;
    die "$filename did not return true value" unless $result;
    $INC{$filename} = $realfilename;
    return $result;
}
```

Note that the file will not be included twice under the same specified name. The file must return TRUE as the last statement to indicate successful execution of any initialization code, so it's customary to end such a file with "`1;`" unless you're sure it'll return TRUE otherwise. But it's better just to put the "`1;`", in case you add more statements.

If EXPR is a bareword, the require assumes a "*.pm*" extension and replaces "*::*" with "*/*" in the filename for you, to make it easy to load standard modules. This form of loading of modules does not risk altering your namespace.

In other words, if you try this:

```
        require Foo::Bar;     # a splendid bareword
```

The require function will actually look for the "*Foo/Bar.pm*" file in the directories specified in the `@INC` array.

But if you try this:

```
        $class = 'Foo::Bar';
        require $class;       # $class is not a bareword
    #or
        require "Foo::Bar";  # not a bareword because of the ""
```

The require function will look for the "*Foo::Bar*" file in the @INC array and will complain about not finding "*Foo::Bar*" there. In this case you can do:

```
        eval "require $class";
```

For a yet-more-powerful import facility, see the `use` entry elsewhere in this document and the *perlmod* manpage.

reset EXPR

reset Generally used in a `continue` block at the end of a loop to clear variables and reset `??` searches so that they work again. The expression is interpreted as a list of single characters (hyphens allowed for ranges). All variables and arrays beginning with one of those letters are reset to their pristine state. If the expression is omitted, one-match searches (`?pattern?`) are reset to match again. Resets only variables or searches in the current package. Always returns 1. Examples:

```
    reset 'X';          # reset all X variables
    reset 'a-z';        # reset lower case variables
    reset;              # just reset ?? searches
```

Resetting `"A-Z"` is not recommended because you'll wipe out your `@ARGV` and `@INC` arrays and your `%ENV` hash. Resets only package variables— lexical variables are unaffected, but they clean themselves up on scope exit anyway, so you'll probably want to use them instead. See the `my` entry elsewhere in this document.

return EXPR

return Returns from a subroutine, `eval()`, or `do FILE` with the value given in EXPR. Evaluation of EXPR may be in list, scalar, or void context, depending on how the return value will be used, and the context may vary from one execution to the next (see `wantarray()`). If no EXPR is given, returns an empty list in list context, an undefined value in scalar context, or nothing in a void context.

(Note that in the absence of a return, a subroutine, eval, or do FILE will automatically return the value of the last expression evaluated.)

reverse LIST

In list context, returns a list value consisting of the elements of LIST in the opposite order. In scalar context, concatenates the elements of LIST, and returns a string value consisting of those bytes, but in the opposite order.

```
    print reverse <>;           # line tac, last line first
```

```
undef $/;                    # for efficiency of <>
print scalar reverse <>;     # byte tac, last line tsrif
```

This operator is also handy for inverting a hash, although there are some caveats. If a value is duplicated in the original hash, only one of those can be represented as a key in the inverted hash. Also, this has to unwind one hash and build a whole new one, which may take some time on a large hash.

```
%by_name = reverse %by_address;      # Invert the hash
```

rewinddir DIRHANDLE
: Sets the current position to the beginning of the directory for the `readdir()` routine on DIRHANDLE.

rindex STR,SUBSTR,POSITION

rindex STR,SUBSTR
: Works just like index except that it returns the position of the LAST occurrence of SUBSTR in STR. If POSITION is specified, returns the last occurrence at or before that position.

rmdir FILENAME

rmdir
: Deletes the directory specified by FILENAME if that directory is empty. If it succeeds it returns TRUE, otherwise it returns FALSE and sets `$!` (errno). If FILENAME is omitted, uses `$_`.

s///
: The substitution operator. See the *perlop* manpage.

scalar EXPR
: Forces EXPR to be interpreted in scalar context and returns the value of EXPR.

```
@counts = ( scalar @a, scalar @b, scalar @c );
```

There is no equivalent operator to force an expression to be interpolated in list context because it's in practice never needed. If you really wanted to do so, however, you could use the construction `@{[ (some expression) ]}`, but usually a simple `(some expression)` suffices.

seek FILEHANDLE,POSITION,WHENCE
: Sets FILEHANDLE's position, just like the `fseek()` call of `stdio()`. FILEHANDLE may be an expression whose value gives the name of the filehandle. The values for WHENCE are `0` to set the new position to POSITION, `1` to set it to the current position plus POSITION, and `2` to set it to EOF plus POSITION (typically negative). For WHENCE you may use the constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` from either the `IO::Seekable` or the POSIX module. Returns `1` upon success, `0` otherwise.

If you want to position file for `sysread()` or `syswrite()`, don't use `seek()` -- buffering makes its effect on the file's system position unpredictable and non-portable. Use `sysseek()` instead.

On some systems you have to do a seek whenever you switch between reading and writing. Amongst other things, this may have the effect of calling stdio's *clearerr*(3). A WHENCE of `1` (`SEEK_CUR`) is useful for not moving the file position:

```
seek(TEST,0,1);
```

This is also useful for applications emulating `tail -f`. Once you hit EOF on your read, and then sleep for a while, you might have to stick in a *seek()* to reset things. The `seek()` doesn't change the current position, but it *does* clear the end-of-file condition on the handle, so that the next `<FILE>` makes Perl try again to read something. We hope.

If that doesn't work (some stdios are particularly cantankerous), then you may need something more like this:

```
for (;;) {
    for ($curpos = tell(FILE); $_ = <FILE>;
         $curpos = tell(FILE)) {
        # search for some stuff and put it into files
    }
    sleep($for_a_while);
    seek(FILE, $curpos, 0);
}
```

seekdir DIRHANDLE,POS

Sets the current position for the `readdir()` routine on DIRHANDLE. POS must be a value returned by `telldir()`. Has the same caveats about possible directory compaction as the corresponding system library routine.

select FILEHANDLE

select

Returns the currently selected filehandle. Sets the current default filehandle for output, if FILEHANDLE is supplied. This has two effects: first, a `write()` or a `print()` without a filehandle will default to this FILEHANDLE. Second, references to variables related to output will refer to this output channel. For example, if you have to set the top of form format for more than one output channel, you might do the following:

```
select(REPORT1);
$^ = 'report1_top';
select(REPORT2);
$^ = 'report2_top';
```

FILEHANDLE may be an expression whose value gives the name of the actual filehandle. Thus:

```
$oldfh = select(STDERR); $| = 1; select($oldfh);
```

Some programmers may prefer to think of filehandles as objects with methods, preferring to write the last example as:

```
use IO::Handle;
STDERR->autoflush(1);
```

select RBITS,WBITS,EBITS,TIMEOUT

This calls the *select*(2) system call with the bit masks specified, which can be constructed using `fileno()` and `vec()`, along these lines:

```
$rin = $win = $ein = '';
vec($rin,fileno(STDIN),1) = 1;
vec($win,fileno(STDOUT),1) = 1;
$ein = $rin | $win;
```

If you want to select on many filehandles you might wish to write a subroutine:

```
sub fhbits {
    my(@fhlist) = split(' ',$_[0]);
    my($bits);
    for (@fhlist) {
        vec($bits,fileno($_),1) = 1;
    }
    $bits;
}
$rin = fhbits('STDIN TTY SOCK');
```

The usual idiom is:

```
($nfound,$timeleft) =
  select($rout=$rin, $wout=$win, $eout=$ein, $timeout);
```

or to block until something becomes ready just do this

```
$nfound = select($rout=$rin, $wout=$win, $eout=$ein, undef);
```

Most systems do not bother to return anything useful in `$timeleft`, so calling *select()* in scalar context just returns `$nfound`.

Any of the bit masks can also be undef. The timeout, if specified, is in seconds, which may be fractional. Note: not all implementations are capable of returning the`$timeleft`. If not, they always return `$timeleft` equal to the supplied `$timeout`.

You can effect a sleep of 250 milliseconds this way:

```
select(undef, undef, undef, 0.25);
```

**WARNING**: One should not attempt to mix buffered I/O (like `read()` or <FH>) with `select()`, except as permitted by POSIX, and even then only on POSIX systems. You have to use `sysread()` instead.

semctl ID,SEMNUM,CMD,ARG

Calls the System V IPC function `semctl()`. You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If CMD is IPC_STAT or GETALL, then ARG must be a variable which will hold the returned semid_ds structure or semaphore value array. Returns like `ioctl()`: the undefined value for error, "`0` but true" for zero, or the actual return value otherwise. See also `IPC::SysV` and `IPC::Semaphore` documentation.

semget KEY,NSEMS,FLAGS

Calls the System V IPC function semget. Returns the semaphore id, or the undefined value if there is an error. See also `IPC::SysV` and `IPC::SysV::Semaphore` documentation.

semop KEY,OPSTRING

Calls the System V IPC function semop to perform semaphore operations such as signaling and waiting. OPSTRING must be a packed array of semop structures. Each semop structure can be generated with `pack("sss", $semnum, $semop, $semflag)`. The number of semaphore operations is implied by the length of OPSTRING. Returns TRUE if successful, or FALSE if there is an error. As an example, the following code waits on semaphore `$semnum` of semaphore id `$semid`:

```
$semop = pack("sss", $semnum, -1, 0);
die "Semaphore trouble: $!\n" unless semop($semid, $semop);
```

To signal the semaphore, replace `-1` with `1`. See also `IPC::SysV` and

`IPC::SysV::Semaphore` documentation.

send SOCKET,MSG,FLAGS,TO

send SOCKET,MSG,FLAGS
Sends a message on a socket. Takes the same flags as the system call of the same name. On unconnected sockets you must specify a destination to send TO, in which case it does a C `sendto()`. Returns the number of characters sent, or the undefined value if there is an error. See the section on *UDP: Message Passing* in the *perlipc* manpage for examples.

setpgrp PID,PGRP
Sets the current process group for the specified PID, `0` for the current process. Will produce a fatal error if used on a machine that doesn't implement *setpgrp*(2). If the arguments are omitted, it defaults to `0,0`. Note that the POSIX version of `setpgrp()` does not accept any arguments, so only setpgrp `0,0` is portable.

setpriority WHICH,WHO,PRIORITY
Sets the current priority for a process, a process group, or a user. (See *setpriority*(2).) Will produce a fatal error if used on a machine that doesn't implement *setpriority*(2).

setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL
Sets the socket option requested. Returns undefined if there is an error. OPTVAL may be specified as `undef` if you don't want to pass an argument.

shift ARRAY

shift
Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If there are no elements in the array, returns the undefined value. If ARRAY is omitted, shifts the `@_` array within the lexical scope of subroutines and formats, and the `@ARGV` array at file scopes or within the lexical scopes established by the `eval ''`, `BEGIN {}`, `END {}`, and `INIT {}` constructs. See also `unshift()`, `push()`, and `pop()`. `Shift()` and `unshift()` do the same thing to the left end of an array that `pop()` and `push()` do to the right end.

shmctl ID,CMD,ARG
Calls the System V IPC function shmctl. You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If CMD is `IPC_STAT`, then ARG must be a variable which will hold the returned `shmid_ds` structure. Returns like ioctl: the undefined value for error, "`0` but true" for zero, or the actual return value otherwise. See also `IPC::SysV` documentation.

shmget KEY,SIZE,FLAGS
Calls the System V IPC function shmget. Returns the shared memory segment id, or the undefined value if there is an error. See also `IPC::SysV` documentation.

shmread ID,VAR,POS,SIZE

shmwrite ID,STRING,POS,SIZE
Reads or writes the System V shared memory segment ID starting at position POS for size SIZE by attaching to it, copying in/out, and detaching from it. When reading, VAR must be a variable that will hold the data read. When writing, if STRING is too long, only SIZE bytes are used; if STRING is too short, nulls are written to fill out SIZE bytes. Return TRUE if successful, or FALSE if there is an error. See also `IPC::SysV` documentation.

shutdown SOCKET,HOW
Shuts down a socket connection in the manner indicated by HOW, which has the same interpretation as in the system call of the same name.

```
shutdown(SOCKET, 0);    # I/we have stopped reading data
shutdown(SOCKET, 1);    # I/we have stopped writing data
shutdown(SOCKET, 2);    # I/we have stopped using this socket
```

This is useful with sockets when you want to tell the other side you're done writing but not done reading, or vice versa. It's also a more insistent form of close because it also disables the filedescriptor in any forked copies in other processes.

sin EXPR

sin Returns the sine of EXPR (expressed in radians). If EXPR is omitted, returns sine of `$_`.

For the inverse sine operation, you may use the `POSIX::asin()` function, or use this relation:

```
sub asin { atan2($_[0], sqrt(1 - $_[0] * $_[0])) }
```

sleep EXPR

sleep Causes the script to sleep for EXPR seconds, or forever if no EXPR. May be interrupted if the process receives a signal such as `SIGALRM`. Returns the number of seconds actually slept. You probably cannot mix `alarm()` and `sleep()` calls, because `sleep()` is often implemented using `alarm()`.

On some older systems, it may sleep up to a full second less than what you requested, depending on how it counts seconds. Most modern systems always sleep the full amount. They may appear to sleep longer than that, however, because your process might not be scheduled right away in a busy multitasking system.

For delays of finer granularity than one second, you may use Perl's `syscall()` interface to access *setitimer*(2) if your system supports it, or else see the `select()` entry elsewhere in this documentabove.

See also the POSIX module's `sigpause()` function.

socket SOCKET,DOMAIN,TYPE,PROTOCOL

Opens a socket of the specified kind and attaches it to filehandle SOCKET. DOMAIN, TYPE, and PROTOCOL are specified the same as for the system call of the same name. You should "`use Socket;`" first to get the proper definitions imported. See the example in the section on *Sockets: Client/Server Communication* in the *perlipc* manpage.

socketpair SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL

Creates an unnamed pair of sockets in the specified domain, of the specified type. DOMAIN, TYPE, and PROTOCOL are specified the same as for the system call of the same name. If unimplemented, yields a fatal error. Returns TRUE if successful.

Some systems defined `pipe()` in terms of `socketpair()`, in which a call to `pipe(Rdr, Wtr)` is essentially:

```
use Socket;
socketpair(Rdr, Wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC);
shutdown(Rdr, 1);        # no more writing for reader
shutdown(Wtr, 0);        # no more reading for writer
```

See the *perlipc* manpage for an example of socketpair use.

sort SUBNAME LIST

sort BLOCK LIST

sort LIST

Sorts the LIST and returns the sorted list value. If SUBNAME or BLOCK is omitted, `sort()`s in standard string comparison order. If SUBNAME is specified, it gives the name of a subroutine that returns an integer less than, equal to, or greater than `0`, depending on how the elements of the array are to be ordered. (The `<=>` and `cmp` operators are extremely useful in such routines.) SUBNAME may be a scalar variable name (unsubscripted), in which case the value provides the name of (or a reference to) the actual subroutine to use. In place of a SUBNAME, you can provide a BLOCK as an anonymous, in-line sort subroutine.

In the interests of efficiency the normal calling code for subroutines is bypassed, with the following effects: the subroutine may not be a recursive subroutine, and the two elements to be compared are passed into the subroutine not via `@_` but as the package global variables `$a` and `$b` (see example below). They are passed by reference, so don't modify `$a` and `$b`. And don't try to declare them as lexicals either.

You also cannot exit out of the sort block or subroutine using any of the loop control operators described in the *perlsyn* manpage or with `goto()`.

When `use locale` is in effect, `sort LIST` sorts LIST according to the current collation locale. See the *perllocale* manpage.

Examples:

```
# sort lexically
@articles = sort @files;

# same thing, but with explicit sort routine
@articles = sort {$a cmp $b} @files;

# now case-insensitively
@articles = sort {uc($a) cmp uc($b)} @files;

# same thing in reversed order
@articles = sort {$b cmp $a} @files;

# sort numerically ascending
@articles = sort {$a <=> $b} @files;

# sort numerically descending
@articles = sort {$b <=> $a} @files;

# sort using explicit subroutine name
sub byage {
    $age{$a} <=> $age{$b};  # presuming numeric
}
@sortedclass = sort byage @class;

# this sorts the %age hash by value instead of key
# using an in-line function
@eldest = sort { $age{$b} <=> $age{$a} } keys %age;
```

```
sub backwards { $b cmp $a; }
@harry = ('dog','cat','x','Cain','Abel');
@george = ('gone','chased','yz','Punished','Axed');
print sort @harry;
        # prints AbelCaincatdogx
print sort backwards @harry;
        # prints xdogcatCainAbel
print sort @george, 'to', @harry;
        # prints AbelAxedCainPunishedcatchaseddoggonetoxyz

# inefficiently sort by descending numeric compare using
# the first integer after the first = sign, or the
# whole record case-insensitively otherwise

@new = sort {
    ($b =~ /=(\d+)/)[0] <=> ($a =~ /=(\d+)/)[0]
                        ||
                uc($a)  cmp  uc($b)
} @old;

# same thing, but much more efficiently;
# we'll build auxiliary indices instead
# for speed
@nums = @caps = ();
for (@old) {
    push @nums, /=(\d+)/;
    push @caps, uc($_);
}

@new = @old[ sort {
                    $nums[$b] <=> $nums[$a]
                             ||
                    $caps[$a] cmp $caps[$b]
                   } 0..$#old
           ];

# same thing using a Schwartzian Transform (no temps)
@new = map { $_->[0] }
    sort { $b->[1] <=> $a->[1]
                    ||
           $a->[2] cmp $b->[2]
    } map { [$_, /=(\d+)/, uc($_)] } @old;
```

If you're using strict, you *MUST NOT* declare `$a` and `$b` as lexicals. They are package globals. That means if you're in the `main` package, it's

```
@articles = sort {$main::b <=> $main::a} @files;
```

or just

```
@articles = sort {$::b <=> $::a} @files;
```

but if you're in the `FooPack` package, it's

```
@articles = sort {$FooPack::b <=> $FooPack::a} @files;
```

The comparison function is required to behave. If it returns inconsistent results (sometimes saying `$x[1]` is less than `$x[2]` and sometimes saying the opposite, for example) the results are not well-defined.

splice ARRAY,OFFSET,LENGTH,LIST

splice ARRAY,OFFSET,LENGTH

splice ARRAY,OFFSET

Removes the elements designated by OFFSET and LENGTH from an array, and replaces them with the elements of LIST, if any. In list context, returns the elements removed from the array. In scalar context, returns the last element removed, or `undef` if no elements are removed. The array grows or shrinks as necessary. If OFFSET is negative then it start that far from the end of the array. If LENGTH is omitted, removes everything from OFFSET onward. If LENGTH is negative, leave that many elements off the end of the array. The following equivalences hold (assuming `$[ == 0`):

```
push(@a,$x,$y)       splice(@a,@a,0,$x,$y)
pop(@a)              splice(@a,-1)
shift(@a)            splice(@a,0,1)
unshift(@a,$x,$y)    splice(@a,0,0,$x,$y)
$a[$x] = $y          splice(@a,$x,1,$y)
```

Example, assuming array lengths are passed before arrays:

```
sub aeq {   # compare two list values
    my(@a) = splice(@_,0,shift);
    my(@b) = splice(@_,0,shift);
    return 0 unless @a == @b;       # same len?
    while (@a) {
        return 0 if pop(@a) ne pop(@b);
    }
    return 1;
}
if (&aeq($len,@foo[1..$len],0+@bar,@bar)) { ... }
```

split /PATTERN/,EXPR,LIMIT

split /PATTERN/,EXPR

split /PATTERN/

split

Splits a string into an array of strings, and returns it. By default, empty leading fields are preserved, and empty trailing ones are deleted.

If not in list context, returns the number of fields found and splits into the `@_` array. (In list context, you can force the split into `@_` by using `??` as the pattern delimiters, but it still returns the list value.) The use of implicit split to `@_` is deprecated, however, because it clobbers your subroutine arguments.

If EXPR is omitted, splits the `$_` string. If PATTERN is also omitted, splits on whitespace (after skipping any leading whitespace). Anything matching PATTERN is taken to be a delimiter separating the fields. (Note that the delimiter may be longer than one character.)

If LIMIT is specified and positive, splits into no more than that many fields (though it may split into fewer). If LIMIT is unspecified or zero, trailing null fields are stripped (which potential users of `pop()` would do well to remember). If LIMIT is negative, it is treated as if an arbitrarily large LIMIT had been specified.

A pattern matching the null string (not to be confused with a null pattern `//`, which is just one member of the set of patterns matching a null string) will split the value of EXPR into separate characters at each point it matches that way. For example:

```
print join(':', split(/ */, 'hi there'));
```

produces the output 'h:i:t:h:e:r:e'.

The LIMIT parameter can be used to split a line partially

```
($login, $passwd, $remainder) = split(/:/, $_, 3);
```

When assigning to a list, if LIMIT is omitted, Perl supplies a LIMIT one larger than the number of variables in the list, to avoid unnecessary work. For the list above LIMIT would have been 4 by default. In time critical applications it behooves you not to split into more fields than you really need.

If the PATTERN contains parentheses, additional array elements are created from each matching substring in the delimiter.

```
split(/([,-])/, "1-10,20", 3);
```

produces the list value

```
(1, '-', 10, ',', 20)
```

If you had the entire header of a normal Unix email message in `$header`, you could split it up into fields and their values this way:

```
$header =~ s/\n\s+/ /g;  # fix continuation lines
%hdrs   =  (UNIX_FROM => split /^(\S*?):\s*/m, $header);
```

The pattern `/PATTERN/` may be replaced with an expression to specify patterns that vary at runtime. (To do runtime compilation only once, use `/$variable/o`.)

As a special case, specifying a PATTERN of space (`' '`) will split on white space just as `split()` with no arguments does. Thus, `split(' ')` can be used to emulate **awk**'s default behavior, whereas `split(/ /)` will give you as many null initial fields as there are leading spaces. A `split()` on `/\s+/` is like a `split(' ')` except that any leading whitespace produces a null first field. A `split()` with no arguments really does a `split(' ', $_)` internally.

Example:

```
open(PASSWD, '/etc/passwd');
while (<PASSWD>) {
     ($login, $passwd, $uid, $gid,
      $gcos, $home, $shell) = split(/:/);
     #...
}
```

(Note that `$shell` above will still have a newline on it. See the `chop`, `chomp`, and `join` entries elsewhere in this document.)

sprintf FORMAT, LIST

Returns a string formatted by the usual `printf()` conventions of the C library function `sprintf()`. See the *sprintf(3)* manpage or the *printf(3)* manpage on your system for an explanation of the general principles.

Perl does its own `sprintf()` formatting -- it emulates the C function `sprintf()`, but it doesn't use it (except for floating-point numbers, and even then only the standard modifiers are allowed). As a result, any non-standard extensions in your local `sprintf()` are not available from Perl.

Perl's `sprintf()` permits the following universally-known conversions:

```
%%   a percent sign
%c   a character with the given number
%s   a string
%d   a signed integer, in decimal
%u   an unsigned integer, in decimal
%o   an unsigned integer, in octal
%x   an unsigned integer, in hexadecimal
%e   a floating-point number, in scientific notation
%f   a floating-point number, in fixed decimal notation
%g   a floating-point number, in %e or %f notation
```

In addition, Perl permits the following widely-supported conversions:

```
%X   like %x, but using upper-case letters
%E   like %e, but using an upper-case "E"
%G   like %g, but with an upper-case "E" (if applicable)
%p   a pointer (outputs the Perl value's address in hexadecimal)
%n   special: *stores* the number of characters output so far
     into the next variable in the parameter list
```

Finally, for backward (and we do mean "backward") compatibility, Perl permits these unnecessary but widely-supported conversions:

```
%i   a synonym for %d
%D   a synonym for %ld
%U   a synonym for %lu
%O   a synonym for %lo
%F   a synonym for %f
```

Perl permits the following universally-known flags between the `%` and the conversion letter:

```
space   prefix positive number with a space
+       prefix positive number with a plus sign
-       left-justify within the field
0       use zeros, not spaces, to right-justify
#       prefix non-zero octal with "0", non-zero hex with "0x"
number  minimum field width
.number "precision": digits after decimal point for
        floating-point, max length for string, minimum length
        for integer
l       interpret integer as C type "long" or "unsigned long"
h       interpret integer as C type "short" or "unsigned short"
```

There is also one Perl-specific flag:

```
V       interpret integer as Perl's standard integer type
```

Where a number would appear in the flags, an asterisk ("`*`") may be used instead, in which case Perl uses the next item in the parameter list as the given number (that is, as the field width or precision). If a field width obtained through "`*`" is negative, it has the same effect as the "`-`" flag: left-justification.

If `use locale` is in effect, the character used for the decimal point in formatted real numbers is affected by the LC_NUMERIC locale. See the *perllocale* manpage.

sqrt EXPR

sqrt Return the square root of EXPR. If EXPR is omitted, returns square root of `$_`.

srand EXPR

srand Sets the random number seed for the `rand()` operator. If EXPR is omitted, uses a semi-random value based on the current time and process ID, among other things. In versions of Perl prior to 5.004 the default seed was just the current `time()`. This isn't a particularly good seed, so many old programs supply their own seed value (often `time ^ $$` or `time ^ ($$ + ($$ << 15))`), but that isn't necessary any more.

In fact, it's usually not necessary to call `srand()` at all, because if it is not called explicitly, it is called implicitly at the first use of the `rand()` operator. However, this was not the case in version of Perl before 5.004, so if your script will run under older Perl versions, it should call `srand()`.

Note that you need something much more random than the default seed for cryptographic purposes. Checksumming the compressed output of one or more rapidly changing operating system status programs is the usual method. For example:

```
srand (time ^ $$ ^ unpack "%L*", `ps axww | gzip`);
```

If you're particularly concerned with this, see the `Math::TrulyRandom` module in CPAN.

Do *not* call `srand()` multiple times in your program unless you know exactly what you're doing and why you're doing it. The point of the function is to "seed" the `rand()` function so that `rand()` can produce a different sequence each time you run your program. Just do it once at the top of your program, or you *won't* get random numbers out of `rand()`!

Frequently called programs (like CGI scripts) that simply use

```
time ^ $$
```

for a seed can fall prey to the mathematical property that

```
a^b == (a+1)^(b+1)
```

one-third of the time. So don't do that.

stat FILEHANDLE

stat EXPR

stat Returns a 13-element list giving the status info for a file, either the file opened via FILEHANDLE, or named by EXPR. If EXPR is omitted, it stats `$_`. Returns a null list if the stat fails. Typically used as follows:

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
   $atime,$mtime,$ctime,$blksize,$blocks)
       = stat($filename);
```

Not all fields are supported on all filesystem types. Here are the meaning of the fields:

```
 0 dev      device number of filesystem
 1 ino      inode number
 2 mode     file mode  (type and permissions)
 3 nlink    number of (hard) links to the file
 4 uid      numeric user ID of file's owner
 5 gid      numeric group ID of file's owner
 6 rdev     the device identifier (special files only)
 7 size     total size of file, in bytes
 8 atime    last access time since the epoch
 9 mtime    last modify time since the epoch
10 ctime    inode change time (NOT creation time!) since the epoch
11 blksize  preferred block size for file system I/O
12 blocks   actual number of blocks allocated
```

(The epoch was at 00:00 January 1, 1970 GMT.)

If stat is passed the special filehandle consisting of an underline, no stat is done, but the current contents of the stat structure from the last stat or filetest are returned. Example:

```
if (-x $file && (($d) = stat(_)) && $d < 0) {
    print "$file is executable NFS file\n";
}
```

(This works on machines only for which the device number is negative under NFS.)

In scalar context, `stat()` returns a boolean value indicating success or failure, and, if successful, sets the information associated with the special filehandle _.

study SCALAR

study Takes extra time to study SCALAR (`$_` if unspecified) in anticipation of doing many pattern matches on the string before it is next modified. This may or may not save time, depending on the nature and number of patterns you are searching on, and on the distribution of character frequencies in the string to be searched -- you probably want to compare run times with and without it to see which runs faster. Those loops which scan for many short constant strings (including the constant parts of more complex patterns) will benefit most. You may have only one `study()` active at a time -- if you study a different scalar the first is "unstudied". (The way `study()` works is this: a linked list of every character in the string to be searched is made, so we know, for example, where all the `'k'` characters are. From each search string, the rarest character is selected, based on some static frequency tables constructed from some C programs and English text. Only those places that contain this "rarest" character are examined.)

For example, here is a loop that inserts index producing entries before any line containing a certain pattern:

```
while (<>) {
    study;
    print ".IX foo\n" if /\bfoo\b/;
    print ".IX bar\n" if /\bbar\b/;
    print ".IX blurfl\n" if /\bblurfl\b/;
    # ...
    print;
}
```

In searching for `/\bfoo\b/`, only those locations in `$_` that contain `"f"` will be looked at, because `"f"` is rarer than `"o"`. In general, this is a big win except in pathological cases. The only question is whether it saves you more time than it took to build the linked list in the first place.

Note that if you have to look for strings that you don't know till runtime, you can build an entire loop as a string and `eval()` that to avoid recompiling all your patterns all the time. Together with undefining `$/` to input entire files as one record, this can be very fast, often faster than specialized programs like *fgrep* (1). The following scans a list of files (`@files`) for a list of words (`@words`), and prints out the names of those files that contain a match:

```
$search = 'while (<>) { study;';
foreach $word (@words) {
    $search .= "++\$seen{\$ARGV} if /\\b$word\\b/;\n";
}
$search .= "}";
@ARGV = @files;
undef $/;
eval $search;                # this screams
$/ = "\n";          # put back to normal input delimiter
foreach $file (sort keys(%seen)) {
    print $file, "\n";
}
```

sub BLOCK

sub NAME

sub NAME BLOCK

This is subroutine definition, not a real function *per se*. With just a NAME (and possibly prototypes), it's just a forward declaration. Without a NAME, it's an anonymous function declaration, and does actually return a value: the CODE ref of the closure you just created. See the *perlsub* manpage and the *perlref* manpage for details.

substr EXPR,OFFSET,LEN,REPLACEMENT

substr EXPR,OFFSET,LEN

substr EXPR,OFFSET

Extracts a substring out of EXPR and returns it. First character is at offset `0`, or whatever you've set `$[` to (but don't do that). If OFFSET is negative (or more precisely, less than `$[`), starts that far from the end of the string. If LEN is omitted, returns everything to the end of the string. If LEN is negative, leaves that many characters off the end of the string.

If you specify a substring that is partly outside the string, the part within the string is returned. If the substring is totally outside the string a warning is produced.

You can use the `substr()` function as an lvalue, in which case EXPR must be an lvalue. If you assign something shorter than LEN, the string will shrink, and if you assign something longer than LEN, the string will grow to accommodate it. To keep the string the same length you may need to pad or chop your value using `sprintf()`.

An alternative to using `substr()` as an lvalue is to specify the replacement string as the 4th argument. This allows you to replace parts of the EXPR and return what was there before in one operation.

symlink OLDFILE,NEWFILE

Creates a new filename symbolically linked to the old filename. Returns `1` for success, `0` otherwise. On systems that don't support symbolic links, produces a fatal error at run time. To check for that, use eval:

```
$symlink_exists =  eval { symlink("",""); 1 };
```

syscall LIST

Calls the system call specified as the first element of the list, passing the remaining elements as arguments to the system call. If unimplemented, produces a fatal error. The arguments are interpreted as follows: if a given argument is numeric, the argument is passed as an int. If not, the pointer to the string value is passed. You are responsible to make sure a string is pre-extended long enough to receive any result that might be written into a string. You can't use a string literal (or other read-only string) as an argument to `syscall()` because Perl has to assume that any string pointer might be written through. If your integer arguments are not literals and have never been interpreted in a numeric context, you may need to add `0` to them to force them to look like numbers. This emulates the `syswrite()` function (or vice versa):

```
require 'syscall.ph';                # may need to run h2ph
$s = "hi there\n";
syscall(&SYS_write, fileno(STDOUT), $s, length $s);
```

Note that Perl supports passing of up to only 14 arguments to your system call, which in practice should usually suffice.

Syscall returns whatever value returned by the system call it calls. If the system call fails, `syscall()` returns `-1` and sets `$!` (errno). Note that some system calls can legitimately return `-1`. The proper way to handle such calls is to assign `$!=0;` before the call and check the value of `$!` if syscall returns `-1`.

There's a problem with `syscall(&SYS_pipe)`: it returns the file number of the read end of the pipe it creates. There is no way to retrieve the file number of the other end. You can avoid this problem by using `pipe()` instead.

sysopen FILEHANDLE,FILENAME,MODE

sysopen FILEHANDLE,FILENAME,MODE,PERMS

Opens the file whose filename is given by FILENAME, and associates it with FILEHANDLE. If FILEHANDLE is an expression, its value is used as the name of the real filehandle wanted. This function calls the underlying operating system's `open()` function with the parameters FILENAME, MODE, PERMS.

The possible values and flag bits of the MODE parameter are system-dependent; they are available via the standard module `Fcntl`. However, for historical reasons, some values are universal: zero means read-only, one means write-only, and two means read/write.

If the file named by FILENAME does not exist and the `open()` call creates it (typically because MODE includes the `O_CREAT` flag), then the value of PERMS specifies the permissions of the newly created file. If you omit the PERMS argument to `sysopen()`, Perl uses the octal value `0666`. These permission values need to be in octal, and are modified by your process's current `umask`. The `umask` value is a number representing disabled permissions bits—if your `umask` were `027` (group can't write; others can't read, write, or execute), then passing `sysopen()` `0666` would create a file with mode `0640` (`0666 &~ 027` is `0640`).

If you find this `umask()` talk confusing, here's some advice: supply a creation mode of `0666` for regular files and one of `0777` for directories (in `mkdir()`) and executable files. This gives users the freedom of choice: if they want protected files, they might choose process umasks of `022`, `027`, or even the particularly antisocial mask of `077`. Programs should rarely if ever make policy decisions better left to the user. The exception to this is when writing files that should be kept private: mail files, web browser cookies, *.rhosts* files, and so on. In short, seldom if ever use `0644` as argument to `sysopen()` because that takes away the user's option to have a more permissive umask. Better to omit it.

The `IO::File` module provides a more object-oriented approach, if you're into that kind of thing.

sysread FILEHANDLE,SCALAR,LENGTH,OFFSET

sysread FILEHANDLE,SCALAR,LENGTH

Attempts to read LENGTH bytes of data into variable SCALAR from the specified FILEHANDLE, using the system call *read*(2). It bypasses stdio, so mixing this with other kinds of reads, `print()`, `write()`, `seek()`, or `tell()` can cause confusion because stdio usually buffers data. Returns the number of bytes actually read, `0` at end of file, or undef if there was an error. SCALAR will be grown or shrunk so that the last byte actually read is the last byte of the scalar after the read.

An OFFSET may be specified to place the read data at some place in the string other than the beginning. A negative OFFSET specifies placement at that many bytes counting backwards from the end of the string. A positive OFFSET greater than the length of SCALAR results in the string being padded to the required size with `"\0"` bytes before the result of the read is appended.

sysseek FILEHANDLE,POSITION,WHENCE

Sets FILEHANDLE's system position using the system call *lseek*(2). It bypasses stdio, so mixing this with reads (other than `sysread()`), `print()`, `write()`, `seek()`, or `tell()` may cause confusion. FILEHANDLE may be an expression whose value gives the name of the filehandle. The values for WHENCE are `0` to set the new position to POSITION, `1` to set the it to the current position plus POSITION, and `2` to set it to EOF plus POSITION (typically negative). For WHENCE, you may use the constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` from either the `IO::Seekable` or the POSIX module.

Returns the new position, or the undefined value on failure. A position of zero is returned as the string "`0` but true"; thus `sysseek()` returns TRUE on success and FALSE on failure, yet you can still easily determine the new position.

system LIST

system PROGRAM LIST

Does exactly the same thing as "`exec LIST`" except that a fork is done first, and the parent process waits for the child process to complete. Note that argument processing varies depending on the number of arguments. If there is more than one argument in LIST, or if LIST is an array with more than one value, starts the program given by the first element of the list with arguments given by the rest of the list. If there is only one scalar argument, the argument is checked for shell metacharacters, and if there are any, the entire argument is passed to the system's command shell for parsing (this is `/bin/sh -c` on Unix platforms, but varies on other platforms). If there are no shell metacharacters in the argument, it is split into words and passed directly to `execvp()`, which is more efficient.

The return value is the exit status of the program as returned by the `wait()` call. To get the actual exit value divide by 256. See also the `exec` entry elsewhere in this document. This is *NOT* what you want to use to capture the output from a command, for that you should use merely backticks or `qx//`, as described in the section on *'STRING'* in the *perlop* manpage.

Like `exec()`, `system()` allows you to lie to a program about its name if you use the "`system PROGRAM LIST`" syntax. Again, see the `exec` entry elsewhere in this document.

Because `system()` and backticks block `SIGINT` and `SIGQUIT`, killing the program they're running doesn't actually interrupt your program.

```
@args = ("command", "arg1", "arg2");
system(@args) == 0
     or die "system @args failed: $?"
```

You can check all the failure possibilities by inspecting `$?` like this:

```
$exit_value  = $? >> 8;
$signal_num  = $? & 127;
$dumped_core = $? & 128;
```

When the arguments get executed via the system shell, results and return codes will be subject to its quirks and capabilities. See the section on *'STRING'* in the *perlop* manpage and the `exec` entry elsewhere in this documentfor details.

syswrite FILEHANDLE,SCALAR,LENGTH,OFFSET

syswrite FILEHANDLE,SCALAR,LENGTH

Attempts to write LENGTH bytes of data from variable SCALAR to the specified FILEHANDLE, using the system call *write*(2). It bypasses stdio, so mixing this with reads (other than `sysread()`), `print()`, `write()`, `seek()`, or `tell()` may cause confusion because stdio usually buffers data. Returns the number of bytes actually written, or `undef` if there was an error. If the LENGTH is greater than the available data in the SCALAR after the OFFSET, only as much data as is available will be written.

An OFFSET may be specified to write the data from some part of the string other than the beginning. A negative OFFSET specifies writing that many bytes counting backwards from the end of the string. In the case the SCALAR is empty you can use OFFSET but only zero offset.

tell FILEHANDLE

tell Returns the current position for FILEHANDLE. FILEHANDLE may be an expression whose value gives the name of the actual filehandle. If FILEHANDLE is omitted, assumes the file last read.

telldir DIRHANDLE

Returns the current position of the `readdir()` routines on DIRHANDLE. Value may be given to `seekdir()` to access a particular location in a directory. Has the same caveats about possible directory compaction as the corresponding system library routine.

tie VARIABLE,CLASSNAME,LIST

This function binds a variable to a package class that will provide the implementation for the variable. VARIABLE is the name of the variable to be enchanted. CLASSNAME is the name of a class implementing objects of correct type. Any additional arguments are passed to the "`new()`" method of the class (meaning `TIESCALAR`, `TIEARRAY`, or `TIEHASH`). Typically these are arguments such as might be passed to the `dbm_open()` function of C. The object returned by the "`new()`" method is also returned by the `tie()` function, which would be useful if you want to access other methods in CLASSNAME.

Note that functions such as `keys()` and `values()` may return huge lists when used on large objects, like DBM files. You may prefer to use the `each()` function to iterate over such. Example:

```
# print out history file offsets
use NDBM_File;
tie(%HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

A class implementing a hash should have the following methods:

```
TIEHASH classname, LIST
DESTROY this
FETCH this, key
STORE this, key, value
DELETE this, key
EXISTS this, key
FIRSTKEY this
NEXTKEY this, lastkey
```

A class implementing an ordinary array should have the following methods:

```
TIEARRAY classname, LIST
DESTROY this
FETCH this, key
STORE this, key, value
[others TBD]
```

A class implementing a scalar should have the following methods:

```
TIESCALAR classname, LIST
DESTROY this
FETCH this,
STORE this, value
```

Unlike `dbmopen()`, the `tie()` function will not use or require a module for you—you need to do that explicitly yourself. See the *DB_File* manpage or the *Config* module for interesting `tie()` implementations.

For further details see the *perltie* manpage, the section on *tied VARIABLE*.

tied VARIABLE
: Returns a reference to the object underlying VARIABLE (the same value that was originally returned by the `tie()` call that bound the variable to a package.) Returns the undefined value if VARIABLE isn't tied to a package.

time
: Returns the number of non-leap seconds since whatever time the system considers to be the epoch (that's 00:00:00, January 1, 1904 for MacOS, and 00:00:00 UTC, January 1, 1970 for most other systems). Suitable for feeding to `gmtime()` and `localtime()`.

times
: Returns a four-element list giving the user and system times, in seconds, for this process and the children of this process.

```
($user,$system,$cuser,$csystem) = times;
```

tr///
: The transliteration operator. Same as `y///`. See the *perlop* manpage.

truncate FILEHANDLE,LENGTH

truncate EXPR,LENGTH
: Truncates the file opened on FILEHANDLE, or named by EXPR, to the specified length. Produces a fatal error if truncate isn't implemented on your system. Returns TRUE if successful, the undefined value otherwise.

uc EXPR

uc
: Returns an uppercased version of EXPR. This is the internal function implementing the `\U` escape in double-quoted strings. Respects current LC_CTYPE locale if `use locale` in force. See the *perllocale* manpage.

If EXPR is omitted, uses `$_`.

ucfirst EXPR

ucfirst Returns the value of EXPR with the first character uppercased. This is the internal function implementing the `\u` escape in double-quoted strings. Respects current LC_CTYPE locale if `use locale` in force. See the *perllocale* manpage.

If EXPR is omitted, uses `$_`.

umask EXPR

umask Sets the umask for the process to EXPR and returns the previous value. If EXPR is omitted, merely returns the current umask.

If *umask*(2) is not implemented on your system and you are trying to restrict access for *yourself* (i.e., (EXPR & 0700) > 0), produces a fatal error at run time. If *umask*(2) is not implemented and you are not trying to restrict access for yourself, returns `undef`.

Remember that a umask is a number, usually given in octal; it is *not* a string of octal digits. See also the `oct` entry elsewhere in this documentif all you have is a string.

undef EXPR

undef Undefines the value of EXPR, which must be an lvalue. Use only on a scalar value, an array (using "`@`"), a hash (using "`%`"), a subroutine (using "`&`"), or a typeglob (using "<*>"). (Saying `undef $hash{$key}` will probably not do what you expect on most predefined variables or DBM list values, so don't do that; see the *delete* manpage.) Always returns the undefined value. You can omit the EXPR, in which case nothing is undefined, but you still get an undefined value that you could, for instance, return from a subroutine, assign to a variable or pass as a parameter. Examples:

```
undef $foo;
undef $bar{'blurfl'};      # Compare to: delete $bar{'blurfl'};
undef @ary;
undef %hash;
undef &mysub;
undef *xyz;       # destroys $xyz, @xyz, %xyz, &xyz, etc.
return (wantarray ? (undef, $errmsg) : undef) if $they_blew_it;
select undef, undef, undef, 0.25;
($a, $b, undef, $c) = &foo;       # Ignore third value returned
```

Note that this is a unary operator, not a list operator.

unlink LIST

unlink Deletes a list of files. Returns the number of files successfully deleted.

```
$cnt = unlink 'a', 'b', 'c';
unlink @goners;
unlink <*.bak>;
```

Note: `unlink()` will not delete directories unless you are superuser and the **–U** flag is supplied to Perl. Even if these conditions are met, be warned that unlinking a directory can inflict damage on your filesystem. Use `rmdir()` instead.

If LIST is omitted, uses `$_`.

unpack TEMPLATE,EXPR

`Unpack()` does the reverse of `pack()`: it takes a string representing a structure and expands it out into a list value, returning the array value. (In scalar context, it returns merely the first value produced.) The TEMPLATE has the same format as in the `pack()` function. Here's a subroutine that does substring:

```
sub substr {
    my($what,$where,$howmuch) = @_;
    unpack("x$where a$howmuch", $what);
}
```

and then there's

```
sub ordinal { unpack("c",$_[0]); } # same as ord()
```

In addition, you may prefix a field with a %<number> to indicate that you want a <number>−bit checksum of the items instead of the items themselves. Default is a 16-bit checksum. For example, the following computes the same number as the System V sum program:

```
while (<>) {
    $checksum += unpack("%16C*", $_);
}
$checksum %= 65536;
```

The following efficiently counts the number of set bits in a bit vector:

```
$setbits = unpack("%32b*", $selectmask);
```

untie VARIABLE

Breaks the binding between a variable and a package. (See `tie()`.)

unshift ARRAY,LIST

Does the opposite of a `shift()`. Or the opposite of a `push()`, depending on how you look at it. Prepends list to the front of the array, and returns the new number of elements in the array.

```
unshift(ARGV, '-e') unless $ARGV[0] =~ /^-/;
```

Note the LIST is prepended whole, not one element at a time, so the prepended elements stay in the same order. Use `reverse()` to do the reverse.

use Module LIST

use Module

use Module VERSION LIST

use VERSION

Imports some semantics into the current package from the named module, generally by aliasing certain subroutine or variable names into your package. It is exactly equivalent to

```
BEGIN { require Module; import Module LIST; }
```

except that Module *must* be a bareword.

If the first argument to `use` is a number, it is treated as a version number instead of a module name. If the version of the Perl interpreter is less than VERSION, then an error message is printed and Perl exits immediately. This is often useful if you need to check the current Perl version before `use`ing library modules that have changed in incompatible ways from older versions of Perl. (We try not to do this more than we have to.)

The `BEGIN` forces the `require` and `import()` to happen at compile time. The `require` makes sure the module is loaded into memory if it hasn't been yet. The `import()` is not a builtin— it's just an ordinary static method call into the "`Module`" package to tell the module to import the list of features back into the current package. The module can implement its `import()` method any way it likes, though most modules just choose to derive their `import()` method via inheritance from the `Exporter` class that is defined in the `Exporter` module. See the *Exporter* manpage. If no `import()` method can be found then the error is currently silently ignored. This may change to a fatal error in a future version.

If you don't want your namespace altered, explicitly supply an empty list:

```
use Module ();
```

That is exactly equivalent to

```
BEGIN { require Module }
```

If the VERSION argument is present between Module and LIST, then the `use` will call the VERSION method in class Module with the given version as an argument. The default VERSION method, inherited from the Universal class, croaks if the given version is larger than the value of the variable `$Module::VERSION`. (Note that there is not a comma after VERSION!)

Because this is a wide-open interface, pragmas (compiler directives) are also implemented this way. Currently implemented pragmas are:

```
use integer;
use diagnostics;
use sigtrap qw(SEGV BUS);
use strict  qw(subs vars refs);
use subs    qw(afunc blurfl);
```

Some of these these pseudo-modules import semantics into the current block scope (like `strict` or `integer`, unlike ordinary modules, which import symbols into the current package (which are effective through the end of the file).

There's a corresponding "`no`" command that unimports meanings imported by `use`, i.e., it calls `unimport Module LIST` instead of `import()`.

```
no integer;
no strict 'refs';
```

If no `unimport()` method can be found the call fails with a fatal error.

See the *perlmod* manpage for a list of standard modules and pragmas.

utime LIST

Changes the access and modification times on each file of a list of files. The first two elements of the list must be the NUMERICAL access and modification times, in that order. Returns the number of files successfully changed. The inode modification time of each file is set to the current time. This code has the same effect as the "`touch`" command if the files already exist:

```
#!/usr/bin/perl
$now = time;
utime $now, $now, @ARGV;
```

values HASH

Returns a list consisting of all the values of the named hash. (In a scalar context, returns the number of values.) The values are returned in an apparently random order, but it is the same order as either the `keys()` or `each()` function would produce on the same hash. As a side effect, it resets HASH's iterator. See also `keys()`, `each()`, and `sort()`.

vec EXPR,OFFSET,BITS

Treats the string in EXPR as a vector of unsigned integers, and returns the value of the bit field specified by OFFSET. BITS specifies the number of bits that are reserved for each entry in the bit vector. This must be a power of two from 1 to 32. `vec()` may also be assigned to, in which case parentheses are needed to give the expression the correct precedence as in

```
vec($image, $max_x * $x + $y, 8) = 3;
```

Vectors created with `vec()` can also be manipulated with the logical operators |, `&`, and `^`, which will assume a bit vector operation is desired when both operands are strings.

The following code will build up an ASCII string saying `'PerlPerlPerl'`. The comments show the string after each step. Note that this code works in the same way on big-endian or little-endian machines.

```
my $foo = '';
vec($foo,  0, 32) = 0x5065726C;     # 'Perl'
vec($foo,  2, 16) = 0x5065;         # 'PerlPe'
vec($foo,  3, 16) = 0x726C;         # 'PerlPerl'
vec($foo,  8,  8) = 0x50;           # 'PerlPerlP'
vec($foo,  9,  8) = 0x65;           # 'PerlPerlPe'
vec($foo, 20,  4) = 2;              # 'PerlPerlPe'   . "\x02"
vec($foo, 21,  4) = 7;              # 'PerlPerlPer'
                                    # 'r' is "\x72"
vec($foo, 45,  2) = 3;              # 'PerlPerlPer'  . "\x0c"
vec($foo, 93,  1) = 1;              # 'PerlPerlPer'  . "\x2c"
vec($foo, 94,  1) = 1;              # 'PerlPerlPerl'
                                    # 'l' is "\x6c"
```

To transform a bit vector into a string or array of 0's and 1's, use these:

```
$bits = unpack("b*", $vector);
@bits = split(//, unpack("b*", $vector));
```

If you know the exact length in bits, it can be used in place of the `*`.

wait
Waits for a child process to terminate and returns the pid of the deceased process, or `-1` if there are no child processes. The status is returned in `$?`.

waitpid PID,FLAGS
Waits for a particular child process to terminate and returns the pid of the deceased process, or `-1` if there is no such child process. The status is returned in `$?`. If you say

```
use POSIX ":sys_wait_h";
#...
waitpid(-1,&WNOHANG);
```

then you can do a non-blocking wait for any process. Non-blocking wait is available on machines supporting either the *waitpid*(2) or *wait4*(2) system calls. However, waiting for a particular pid with FLAGS of `0` is implemented everywhere. (Perl emulates the system call by remembering the status values of processes that have exited but have not been harvested by the Perl script yet.)

See the *perlipc* manpage for other examples.

wantarray
Returns TRUE if the context of the currently executing subroutine is looking for a list value. Returns FALSE if the context is looking for a scalar. Returns the undefined value if the context is looking for no value (void context).

```
return unless defined wantarray;    # don't bother doing more
my @a = complex_calculation();
return wantarray ? @a : "@a";
```

warn LIST

Produces a message on STDERR just like `die()`, but doesn't exit or throw an exception.

If LIST is empty and `$@` already contains a value (typically from a previous eval) that value is used after appending `"\t...caught"` to `$@`. This is useful for staying almost, but not entirely similar to `die()`.

If `$@` is empty then the string `"Warning: Something's wrong"` is used.

No message is printed if there is a `$SIG{__WARN__}` handler installed. It is the handler's responsibility to deal with the message as it sees fit (like, for instance, converting it into a `die()`). Most handlers must therefore make arrangements to actually display the warnings that they are not prepared to deal with, by calling `warn()` again in the handler. Note that this is quite safe and will not produce an endless loop, since `__WARN__` hooks are not called from inside one.

You will find this behavior is slightly different from that of `$SIG{__DIE__}` handlers (which don't suppress the error text, but can instead call `die()` again to change it).

Using a `__WARN__` handler provides a powerful way to silence all warnings (even the so-called mandatory ones). An example:

```
# wipe out *all* compile-time warnings
BEGIN { $SIG{'__WARN__'} = sub { warn $_[0] if $DOWARN } }
my $foo = 10;
my $foo = 20;          # no warning about duplicate my $foo,
                       # but hey, you asked for it!
# no compile-time or run-time warnings before here
$DOWARN = 1;

# run-time warnings enabled after here
warn "\$foo is alive and $foo!";     # does show up
```

See the *perlvar* manpage for details on setting `%SIG` entries, and for more examples.

write FILEHANDLE

write EXPR

write Writes a formatted record (possibly multi-line) to the specified FILEHANDLE, using the format associated with that file. By default the format for a file is the one having the same name as the filehandle, but the format for the current output channel (see the `select()` function) may be set explicitly by assigning the name of the format to the `$~` variable.

Top of form processing is handled automatically: if there is insufficient room on the current page for the formatted record, the page is advanced by writing a form feed, a special top-of-page format is used to format the new page header, and then the record is written. By default the top-of-page format is the name of the filehandle with "_TOP" appended, but it may be dynamically set to the format of your choice by assigning the name to the `$^` variable while the filehandle is selected. The number of lines remaining on the current page is in variable `$-`, which can be set to `0` to force a new page.

If FILEHANDLE is unspecified, output goes to the current default output channel, which starts out as STDOUT but may be changed by the `select()` operator. If the FILEHANDLE is an EXPR, then the expression is evaluated and the resulting string is used to look up the name of the FILEHANDLE at run time. For more on formats, see the *perlform* manpage.

Note that write is *NOT* the opposite of `read()`. Unfortunately.

y/// The transliteration operator. Same as `tr///`. See the *perlop* manpage.

## NAME

perlvar – Perl predefined variables

## DESCRIPTION

### Predefined Names

The following names have special meaning to Perl. Most punctuation names have reasonable mnemonics, or analogues in one of the shells. Nevertheless, if you wish to use long variable names, you just need to say

```
use English;
```

at the top of your program. This will alias all the short names to the long names in the current package. Some even have medium names, generally borrowed from **awk**.

To go a step further, those variables that depend on the currently selected filehandle may instead (and preferably) be set by calling an object method on the FileHandle object. (Summary lines below for this contain the word HANDLE.) First you must say

```
use FileHandle;
```

after which you may use either

```
method HANDLE EXPR
```

or more safely,

```
HANDLE->method(EXPR)
```

Each of the methods returns the old value of the FileHandle attribute. The methods each take an optional EXPR, which if supplied specifies the new value for the FileHandle attribute in question. If not supplied, most of the methods do nothing to the current value, except for *autoflush()*, which will assume a 1 for you, just to be different.

A few of these variables are considered "read-only". This means that if you try to assign to this variable, either directly or indirectly through a reference, you'll raise a run-time exception.

The following list is ordered by scalar variables first, then the arrays, then the hashes (except $^M was added in the wrong place). This is somewhat obscured by the fact that `%ENV` and `%SIG` are listed as `$ENV`{expr} and `$SIG`{expr}.

$ARG

$_ The default input and pattern-searching space. The following pairs are equivalent:

```
while (<>) {...}     # equivalent in only while!
while (defined($_ = <>)) {...}

/^Subject:/
$_ =~ /^Subject:/

tr/a-z/A-Z/
$_ =~ tr/a-z/A-Z/

chop
chop($_)
```

Here are the places where Perl will assume `$_` even if you don't use it:

- Various unary functions, including functions like *ord()* and *int()*, as well as the all file tests (`-f`, `-d`) except for `-t`, which defaults to STDIN.

- Various list functions like *print()* and *unlink()*.
- The pattern matching operations `m//`, `s///`, and `tr///` when used without an `=~` operator.
- The default iterator variable in a `foreach` loop if no other variable is supplied.
- The implicit iterator variable in the *grep()* and *map()* functions.
- The default place to put an input record when a `<FH>` operation's result is tested by itself as the sole criterion of a `while` test. Note that outside of a `while` test, this will not happen.

  (Mnemonic: underline is understood in certain operations.)

$<*digits*>
Contains the subpattern from the corresponding set of parentheses in the last pattern matched, not counting patterns matched in nested blocks that have been exited already. (Mnemonic: like \digits.) These variables are all read-only.

$MATCH

$& The string matched by the last successful pattern match (not counting any matches hidden within a BLOCK or *eval()* enclosed by the current BLOCK). (Mnemonic: like & in some editors.) This variable is read-only.

$PREMATCH

$` The string preceding whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or eval enclosed by the current BLOCK). (Mnemonic: ` often precedes a quoted string.) This variable is read-only.

$POSTMATCH

$' The string following whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or *eval()* enclosed by the current BLOCK). (Mnemonic: ' often follows a quoted string.) Example:

```
$_ = 'abcdefghi';
/def/;
print "$`:$&:$'\n";          # prints abc:def:ghi
```

This variable is read-only.

$LAST_PAREN_MATCH

$+ The last bracket matched by the last search pattern. This is useful if you don't know which of a set of alternative patterns matched. For example:

```
/Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

(Mnemonic: be positive and forward looking.) This variable is read-only.

$MULTILINE_MATCHING

$* Set to 1 to do multi-line matching within a string, 0 to tell Perl that it can assume that strings contain a single line, for the purpose of optimizing pattern matches. Pattern matches on strings containing multiple newlines can produce confusing results when "`$*`" is 0. Default is 0. (Mnemonic: * matches multiple things.) Note that this variable influences the interpretation of only "`^`" and "`$`". A literal newline can be searched for even when `$* == 0`.

Use of "`$*`" is deprecated in modern Perls, supplanted by the `/s` and `/m` modifiers on pattern matching.

input_line_number HANDLE EXPR

$INPUT_LINE_NUMBER

$NR

$. The current input line number for the last file handle from which you read (or performed a `seek` or `tell` on). An explicit close on a filehandle resets the line number. Because "<>" never does an explicit close, line numbers increase across ARGV files (but see examples under *eof()*). Localizing `$.` has the effect of also localizing Perl's notion of "the last read filehandle". (Mnemonic: many programs use "." to mean the current line number.)

input_record_separator HANDLE EXPR

$INPUT_RECORD_SEPARATOR

$RS

$/ The input record separator, newline by default. Works like **awk**'s RS variable, including treating empty lines as delimiters if set to the null string. (Note: An empty line cannot contain any spaces or tabs.) You may set it to a multi-character string to match a multi-character delimiter, or to `undef` to read to end of file. Note that setting it to `"\n\n"` means something slightly different than setting it to `""`, if the file contains consecutive empty lines. Setting it to `""` will treat two or more consecutive empty lines as a single empty line. Setting it to `"\n\n"` will blindly assume that the next input character belongs to the next paragraph, even if it's a newline. (Mnemonic: / is used to delimit line boundaries when quoting poetry.)

```
undef $/;
$_ = <FH>;          # whole file now here
s/\n[ \t]+/ /g;
```

Remember: the value of $/ is a string, not a regexp. AWK has to be better for something :–)

Setting $/ to a reference to an integer, scalar containing an integer, or scalar that's convertable to an integer will attempt to read records instead of lines, with the maximum record size being the referenced integer. So this:

```
$/ = \32768; # or \"32768", or \$var_containing_32768
open(FILE, $myfile);
$_ = <FILE>;
```

will read a record of no more than 32768 bytes from FILE. If you're not reading from a record-oriented file (or your OS doesn't have record-oriented files), then you'll likely get a full chunk of data with every read. If a record is larger than the record size you've set, you'll get the record back in pieces.

On VMS, record reads are done with the equivalent of `sysread`, so it's best not to mix record and non-record reads on the same file. (This is likely not a problem, as any file you'd want to read in record mode is proably usable in line mode) Non-VMS systems perform normal I/O, so it's safe to mix record and non-record reads of a file.

autoflush HANDLE EXPR

$OUTPUT_AUTOFLUSH

$| If set to nonzero, forces a flush right away and after every write or print on the currently selected output channel. Default is 0 (regardless of whether the channel is actually buffered by the system or not; `$|` tells you only whether you've asked Perl explicitly to flush after each write). Note that STDOUT will typically be line buffered if output is to the terminal and block buffered otherwise. Setting this variable is useful primarily when you are outputting to a pipe, such as when you are running a Perl script under rsh and want to see the output as it's happening. This has no effect on input buffering. (Mnemonic: when you want your pipes to be piping hot.)

output_field_separator HANDLE EXPR

$OUTPUT_FIELD_SEPARATOR

$OFS

$, The output field separator for the print operator. Ordinarily the print operator simply prints out the comma-separated fields you specify. To get behavior more like **awk**, set this variable as you would set **awk**'s OFS variable to specify what is printed between fields. (Mnemonic: what is printed when there is a , in your print statement.)

output_record_separator HANDLE EXPR

$OUTPUT_RECORD_SEPARATOR

$ORS

$\ The output record separator for the print operator. Ordinarily the print operator simply prints out the comma-separated fields you specify, with no trailing newline or record separator assumed. To get behavior more like **awk**, set this variable as you would set **awk**'s ORS variable to specify what is printed at the end of the print. (Mnemonic: you set ``$\`` instead of adding \n at the end of the print. Also, it's just like `$/`, but it's what you get ``back'' from Perl.)

$LIST_SEPARATOR

$" This is like ``$,`` except that it applies to array values interpolated into a double-quoted string (or similar interpreted string). Default is a space. (Mnemonic: obvious, I think.)

$SUBSCRIPT_SEPARATOR

$SUBSEP

$; The subscript separator for multidimensional array emulation. If you refer to a hash element as

```
$foo{$a,$b,$c}
```

it really means

```
$foo{join($;, $a, $b, $c)}
```

But don't put

```
@foo{$a,$b,$c}      # a slice--note the @
```

which means

```
($foo{$a},$foo{$b},$foo{$c})
```

Default is ``\034'', the same as SUBSEP in **awk**. Note that if your keys contain binary data there might not be any safe value for ``$;``. (Mnemonic: comma (the syntactic subscript separator) is a semi-semicolon. Yeah, I know, it's pretty lame, but ``$,`` is already taken for something more important.)

Consider using ``real'' multidimensional arrays.

$OFMT

$# The output format for printed numbers. This variable is a half-hearted attempt to emulate **awk**'s OFMT variable. There are times, however, when **awk** and Perl have differing notions of what is in fact numeric. The initial value is %.*n*g, where *n* is the value of the macro DBL_DIG from your system's *float.h*. This is different from **awk**'s default OFMT setting of %.6g, so you need to set ``$#`` explicitly to get **awk**'s value. (Mnemonic: # is the number sign.)

Use of ``$#`` is deprecated.

format_page_number HANDLE EXPR

$FORMAT_PAGE_NUMBER

$% The current page number of the currently selected output channel. (Mnemonic: % is page number in **nroff**.)

format_lines_per_page HANDLE EXPR

$FORMAT_LINES_PER_PAGE

$= The current page length (printable lines) of the currently selected output channel. Default is 60. (Mnemonic: = has horizontal lines.)

format_lines_left HANDLE EXPR

$FORMAT_LINES_LEFT

$- The number of lines left on the page of the currently selected output channel. (Mnemonic: lines_on_page – lines_printed.)

format_name HANDLE EXPR

$FORMAT_NAME

$~ The name of the current report format for the currently selected output channel. Default is name of the filehandle. (Mnemonic: brother to ``$^''.)

format_top_name HANDLE EXPR

$FORMAT_TOP_NAME

$^ The name of the current top-of-page format for the currently selected output channel. Default is name of the filehandle with _TOP appended. (Mnemonic: points to top of page.)

format_line_break_characters HANDLE EXPR

$FORMAT_LINE_BREAK_CHARACTERS

$: The current set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format. Default is `` \n-'', to break on whitespace or hyphens. (Mnemonic: a ``colon'' in poetry is a part of a line.)

format_formfeed HANDLE EXPR

$FORMAT_FORMFEED

$^L What formats output to perform a form feed. Default is \f.

$ACCUMULATOR

$^A The current value of the *write()* accumulator for *format()* lines. A format contains *formline()* commands that put their result into `$^A`. After calling its format, *write()* prints out the contents of `$^A` and empties. So you never actually see the contents of `$^A` unless you call *formline()* yourself and then look at it. See the *perlform* manpage and the `formline()` entry in the *perlfunc* manpage.

$CHILD_ERROR

$? The status returned by the last pipe close, backtick (``` `` ```) command, or *system()* operator. Note that this is the status word returned by the *wait()* system call (or else is made up to look like it). Thus, the exit value of the subprocess is actually (`$? >> 8`), and `$? & 127` gives which signal, if any, the process died from, and `$? & 128` reports whether there was a core dump. (Mnemonic: similar to **sh** and **ksh**.)

Additionally, if the `h_errno` variable is supported in C, its value is returned via $? if any of the `gethost*()` functions fail.

Note that if you have installed a signal handler for `SIGCHLD`, the value of `$?` will usually be wrong outside that handler.

Inside an END subroutine `$?` contains the value that is going to be given to `exit()`. You can modify `$?` in an END subroutine to change the exit status of the script.

Under VMS, the pragma `use vmsish 'status'` makes `$?` reflect the actual VMS exit status, instead of the default emulation of POSIX status.

Also see the section on *Error Indicators*.

$OS_ERROR

$ERRNO

$! If used in a numeric context, yields the current value of errno, with all the usual caveats. (This means that you shouldn't depend on the value of `$!` to be anything in particular unless you've gotten a specific error return indicating a system error.) If used in a string context, yields the corresponding system error string. You can assign to `$!` to set *errno* if, for instance, you want `"$!"` to return the string for error *n*, or you want to set the exit value for the *die()* operator. (Mnemonic: What just went bang?)

Also see the section on *Error Indicators*.

$EXTENDED_OS_ERROR

$^E Error information specific to the current operating system. At the moment, this differs from `$!` under only VMS, OS/2, and Win32 (and for MacPerl). On all other platforms, `$^E` is always just the same as `$!`.

Under VMS, `$^E` provides the VMS status value from the last system error. This is more specific information about the last system error than that provided by `$!`. This is particularly important when `$!` is set to **EVMSERR**.

Under OS/2, `$^E` is set to the error code of the last call to OS/2 API either via CRT, or directly from perl.

Under Win32, `$^E` always returns the last error information reported by the Win32 call `GetLastError()` which describes the last error from within the Win32 API. Most Win32-specific code will report errors via `$^E`. ANSI C and UNIX–like calls set `errno` and so most portable Perl code will report errors via `$!`.

Caveats mentioned in the description of `$!` generally apply to `$^E`, also. (Mnemonic: Extra error explanation.)

Also see the section on *Error Indicators*.

$EVAL_ERROR

$@ The Perl syntax error message from the last *eval()* command. If null, the last *eval()* parsed and executed correctly (although the operations you invoked may have failed in the normal fashion). (Mnemonic: Where was the syntax error "at"?)

Note that warning messages are not collected in this variable. You can, however, set up a routine to process warnings by setting `$SIG{__WARN__}` as described below.

Also see the section on *Error Indicators*.

$PROCESS_ID

$PID

$$ The process number of the Perl running this script. (Mnemonic: same as shells.)

$REAL_USER_ID

$UID

$< The real uid of this process. (Mnemonic: it's the uid you came *FROM*, if you're running setuid.)

$EFFECTIVE_USER_ID

$EUID

$> The effective uid of this process. Example:

```
$< = $>;            # set real to effective uid
($<,$>) = ($>,$<);  # swap real and effective uid
```

(Mnemonic: it's the uid you went *TO*, if you're running setuid.) Note: "`$<`" and "`$>`" can be swapped only on machines supporting *setreuid()*.

$REAL_GROUP_ID

$GID

$( The real gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by *getgid()*, and the subsequent ones by *getgroups()*, one of which may be the same as the first number.

However, a value assigned to "`$(`" must be a single number used to set the real gid. So the value given by "`$(`" should *not* be assigned back to "`$(`" without being forced numeric, such as by adding zero.

(Mnemonic: parentheses are used to *GROUP* things. The real gid is the group you *LEFT*, if you're running setgid.)

$EFFECTIVE_GROUP_ID

$EGID

$) The effective gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by *getegid()*, and the subsequent ones by *getgroups()*, one of which may be the same as the first number.

Similarly, a value assigned to "`$)`" must also be a space-separated list of numbers. The first number is used to set the effective gid, and the rest (if any) are passed to *setgroups()*. To get the effect of an empty list for *setgroups()*, just repeat the new effective gid; that is, to force an effective gid of 5 and an effectively empty *setgroups()* list, say `$) = "5 5"` .

(Mnemonic: parentheses are used to *GROUP* things. The effective gid is the group that's *RIGHT* for you, if you're running setgid.)

Note: "`$<`", "`$>`", "`$(`" and "`$)`" can be set only on machines that support the corresponding *set[re][ug]id()* routine. "`$(`" and "`$)`" can be swapped only on machines supporting *setregid()*.

$PROGRAM_NAME

$0 Contains the name of the file containing the Perl script being executed. On some operating systems assigning to "`$0`" modifies the argument area that the *ps*(1) program sees. This is more useful as a way of indicating the current program state than it is for hiding the program you're running. (Mnemonic: same as **sh** and **ksh**.)

$[ The index of the first element in an array, and of the first character in a substring. Default is 0, but you could set it to 1 to make Perl behave more like **awk** (or Fortran) when subscripting and when evaluating the *index()* and *substr()* functions. (Mnemonic: [ begins subscripts.)

As of Perl 5, assignment to "`$[`" is treated as a compiler directive, and cannot influence the behavior of any other file. Its use is discouraged.

$PERL_VERSION

$] The version + patchlevel / 1000 of the Perl interpreter. This variable can be used to determine whether the Perl interpreter executing a script is in the right range of versions. (Mnemonic: Is this version of perl in the right bracket?) Example:

```
warn "No checksumming!\n" if $] < 3.019;
```

See also the documentation of `use VERSION` and `require VERSION` for a convenient way to fail if the Perl interpreter is too old.

$DEBUGGING

$^D The current value of the debugging flags. (Mnemonic: value of **–D** switch.)

$SYSTEM_FD_MAX

$^F The maximum system file descriptor, ordinarily 2. System file descriptors are passed to *exec()*ed processes, while higher file descriptors are not. Also, during an *open()*, system file descriptors are preserved even if the *open()* fails. (Ordinary file descriptors are closed before the *open()* is attempted.) Note that the close-on-exec status of a file descriptor will be decided according to the value of `$^F` at the time of the open, not the time of the exec.

$^H The current set of syntax checks enabled by `use strict` and other block scoped compiler hints. See the documentation of `strict` for more details.

$INPLACE_EDIT

$^I The current value of the inplace-edit extension. Use `undef` to disable inplace editing. (Mnemonic: value of **–i** switch.)

$^M By default, running out of memory it is not trappable. However, if compiled for this, Perl may use the contents of `$^M` as an emergency pool after *die()*ing with this message. Suppose that your Perl were compiled with –DPERL_EMERGENCY_SBRK and used Perl's malloc. Then

```
$^M = 'a' x (1<<16);
```

would allocate a 64K buffer for use when in emergency. See the *INSTALL* file for information on how to enable this option. As a disincentive to casual use of this advanced feature, there is no the *English* manpage long name for this variable.

$OSNAME

$^O The name of the operating system under which this copy of Perl was built, as determined during the configuration process. The value is identical to `$Config{'osname'}`.

$PERLDB

$^P The internal variable for debugging support. Different bits mean the following (subject to change):

0x01 Debug subroutine enter/exit.

0x02 Line-by-line debugging.

0x04 Switch off optimizations.

0x08 Preserve more data for future interactive inspections.

0x10 Keep info about source lines on which a subroutine is defined.

0x20 Start with single-step on.

Note that some bits may be relevent at compile-time only, some at run-time only. This is a new mechanism and the details may change.

$^R The result of evaluation of the last successful the section on `(?{ code })` in the *perlre* manpage regular expression assertion. (Excluding those used as switches.) May be written to.

$^S Current state of the interpreter. Undefined if parsing of the current module/eval is not finished (may happen in `$SIG{__DIE__}` and `$SIG{__WARN__}` handlers). True if inside an eval, otherwise false.

$BASETIME

$^T The time at which the script began running, in seconds since the epoch (beginning of 1970). The values returned by the **–M**, **–A**, and **–C** filetests are based on this value.

$WARNING

$^W The current value of the warning switch, either TRUE or FALSE. (Mnemonic: related to the **–w** switch.)

$EXECUTABLE_NAME

$^X The name that the Perl binary itself was executed as, from C's `argv[0]`.

$ARGV contains the name of the current file when reading from <>.

@ARGV The array `@ARGV` contains the command line arguments intended for the script. Note that `$#ARGV` is the generally number of arguments minus one, because `$ARGV[0]` is the first argument, *NOT* the command name. See "`$0`" for the command name.

@INC The array `@INC` contains the list of places to look for Perl scripts to be evaluated by the `do EXPR`, `require`, or `use` constructs. It initially consists of the arguments to any **–I** command line switches, followed by the default Perl library, probably */usr/local/lib/perl*, followed by ".", to represent the current directory. If you need to modify this at runtime, you should use the `use lib` pragma to get the machine-dependent library properly loaded also:

```
use lib '/mypath/libdir/';
use SomeMod;
```

@_ Within a subroutine the array `@_` contains the parameters passed to that subroutine. See the *perlsub* manpage.

%INC The hash `%INC` contains entries for each filename that has been included via `do` or `require`. The key is the filename you specified, and the value is the location of the file actually found. The `require` command uses this array to determine whether a given file has already been included.

%ENV $ENV{expr}
The hash `%ENV` contains your current environment. Setting a value in `ENV` changes the environment for child processes.

%SIG $SIG{expr}
The hash `%SIG` is used to set signal handlers for various signals. Example:

```
sub handler {        # 1st argument is signal name
    my($sig) = @_;
    print "Caught a SIG$sig--shutting down\n";
    close(LOG);
    exit(0);
}

$SIG{'INT'}  = \&handler;
$SIG{'QUIT'} = \&handler;
...
$SIG{'INT'} = 'DEFAULT';     # restore default action
$SIG{'QUIT'} = 'IGNORE';     # ignore SIGQUIT
```

The `%SIG` array contains values for only the signals actually set within the Perl script. Here are some other examples:

```
$SIG{"PIPE"} = Plumber;      # SCARY!!
$SIG{"PIPE"} = "Plumber";    # assumes main::Plumber (not recommended)
$SIG{"PIPE"} = \&Plumber;    # just fine; assume current Plumber
$SIG{"PIPE"} = Plumber();    # oops, what did Plumber() return??
```

The one marked scary is problematic because it's a bareword, which means sometimes it's a string representing the function, and sometimes it's going to call the subroutine call right then and there! Best to be sure and quote it or take a reference to it. *Plumber works too. See the *perlsub* manpage.

If your system has the *sigaction()* function then signal handlers are installed using it. This means you get reliable signal handling. If your system has the SA_RESTART flag it is used when signals handlers are installed. This means that system calls for which it is supported continue rather than returning when a signal arrives. If you want your system calls to be interrupted by signal delivery then do something like this:

```
use POSIX ':signal_h';

my $alarm = 0;
sigaction SIGALRM, new POSIX::SigAction sub { $alarm = 1 }
    or die "Error setting SIGALRM handler: $!\n";
```

See the *POSIX* manpage.

Certain internal hooks can be also set using the `%SIG` hash. The routine indicated by `$SIG{__WARN__}` is called when a warning message is about to be printed. The warning message is passed as the first argument. The presence of a __WARN__ hook causes the ordinary printing of warnings to STDERR to be suppressed. You can use this to save warnings in a variable, or turn warnings into fatal errors, like this:

```
local $SIG{__WARN__} = sub { die $_[0] };
eval $proggie;
```

The routine indicated by `$SIG{__DIE__}` is called when a fatal exception is about to be thrown. The error message is passed as the first argument. When a __DIE__ hook routine returns, the exception processing continues as it would have in the absence of the hook, unless the hook routine itself exits via a `goto`, a loop exit, or a *die()*. The `__DIE__` handler is explicitly disabled during the call, so that you can die from a `__DIE__` handler. Similarly for `__WARN__`.

Note that the `$SIG{__DIE__}` hook is called even inside *eval()*ed blocks/strings. See the `die` entry in the *perlfunc* manpage and the section on *$^S* in the *perlvar* manpage for how to circumvent this.

Note that `__DIE__`/`__WARN__` handlers are very special in one respect: they may be called to report (probable) errors found by the parser. In such a case the parser may be in inconsistent state, so any attempt to evaluate Perl code from such a handler will probably result in a segfault. This means that calls which result/may-result in parsing Perl should be used with extreme causion, like this:

```
require Carp if defined $^S;
Carp::confess("Something wrong") if defined &Carp::confess;
die "Something wrong, but could not load Carp to give backtrace...
     To see backtrace try starting Perl with -MCarp switch";
```

Here the first line will load Carp *unless* it is the parser who called the handler. The second line will print backtrace and die if Carp was available. The third line will be executed only if Carp was not available.

See the `die` entry in the *perlfunc* manpage, the `warn` entry in the *perlfunc* manpage and the `eval` entry in the *perlfunc* manpage for additional info.

**Error Indicators**

The variables the section on *$@*, the section on *$!*, the section on *$^E*, and the section on *$?* contain information about different types of error conditions that may appear during execution of Perl script. The variables are shown ordered by the ''distance'' between the subsystem which reported the error and the Perl process, and correspond to errors detected by the Perl interpreter, C library, operating system, or an external program, respectively.

To illustrate the differences between these variables, consider the following Perl expression:

```
eval '
      open PIPE, "/cdrom/install |";
      @res = <PIPE>;
      close PIPE or die "bad pipe: $?, $!";
     ';
```

After execution of this statement all 4 variables may have been set.

$@ is set if the string to be `eval`–ed did not compile (this may happen if `open` or `close` were imported with bad prototypes), or if Perl code executed during evaluation *die()*d (either implicitly, say, if `open` was imported from module the *Fatal* manpage, or the `die` after `close` was triggered). In these cases the value of $@ is the compile error, or `Fatal` error (which will interpolate `$!`!), or the argument to `die` (which will interpolate `$!` and `$?`!).

When the above expression is executed, *open()*, `<PIPE>`, and `close` are translated to C run-time library calls. $! is set if one of these calls fails. The value is a symbolic indicator chosen by the C run-time library, say `No such file or directory`.

On some systems the above C library calls are further translated to calls to the kernel. The kernel may have set more verbose error indicator that one of the handful of standard C errors. In such cases $^E contains this verbose error indicator, which may be, say, `CDROM tray not closed`. On systems where C library calls are identical to system calls $^E is a duplicate of $!.

Finally, $? may be set to non-`0` value if the external program `/cdrom/install` fails. Upper bits of the particular value may reflect specific error conditions encountered by this program (this is program-dependent), lower-bits reflect mode of failure (segfault, completion, etc.). Note that in contrast to $@, $!, and $^E, which are set only if error condition is detected, the variable $? is set on each `wait` or pipe `close`, overwriting the old value.

For more details, see the individual descriptions at the section on *$@*, the section on *$!*, the section on *$^E*, and the section on *$?*.

## NAME

perlsub – Perl subroutines

## SYNOPSIS

To declare subroutines:

```
sub NAME;              # A "forward" declaration.
sub NAME(PROTO);       #  ditto, but with prototypes

sub NAME BLOCK         # A declaration and a definition.
sub NAME(PROTO) BLOCK #  ditto, but with prototypes
```

To define an anonymous subroutine at runtime:

```
$subref = sub BLOCK;           # no proto
$subref = sub (PROTO) BLOCK;   # with proto
```

To import subroutines:

```
use PACKAGE qw(NAME1 NAME2 NAME3);
```

To call subroutines:

```
NAME(LIST);     # & is optional with parentheses.
NAME LIST;      # Parentheses optional if predeclared/imported.
&NAME;          # Makes current @_ visible to called subroutine.
```

## DESCRIPTION

Like many languages, Perl provides for user-defined subroutines. These may be located anywhere in the main program, loaded in from other files via the `do`, `require`, or `use` keywords, or even generated on the fly using `eval` or anonymous subroutines (closures). You can even call a function indirectly using a variable containing its name or a CODE reference to it.

The Perl model for function call and return values is simple: all functions are passed as parameters one single flat list of scalars, and all functions likewise return to their caller one single flat list of scalars. Any arrays or hashes in these call and return lists will collapse, losing their identities—but you may always use pass-by-reference instead to avoid this. Both call and return lists may contain as many or as few scalar elements as you'd like. (Often a function without an explicit return statement is called a subroutine, but there's really no difference from the language's perspective.)

Any arguments passed to the routine come in as the array `@_`. Thus if you called a function with two arguments, those would be stored in `$_[0]` and `$_[1]`. The array `@_` is a local array, but its elements are aliases for the actual scalar parameters. In particular, if an element `$_[0]` is updated, the corresponding argument is updated (or an error occurs if it is not updatable). If an argument is an array or hash element which did not exist when the function was called, that element is created only when (and if) it is modified or if a reference to it is taken. (Some earlier versions of Perl created the element whether or not it was assigned to.) Note that assigning to the whole array `@_` removes the aliasing, and does not update any arguments.

The return value of the subroutine is the value of the last expression evaluated. Alternatively, a `return` statement may be used to exit the subroutine, optionally specifying the returned value, which will be evaluated in the appropriate context (list, scalar, or void) depending on the context of the subroutine call. If you specify no return value, the subroutine will return an empty list in a list context, an undefined value in a scalar context, or nothing in a void context. If you return one or more arrays and/or hashes, these will be flattened together into one large indistinguishable list.

Perl does not have named formal parameters, but in practice all you do is assign to a `my()` list of these. Any variables you use in the function that aren't declared private are global variables. For the gory details on creating private variables, see the section on *Private Variables via my()* and the section on *Temporary*

*Values via local()*. To create protected environments for a set of functions in a separate package (and probably a separate file), see the section on *Packages* in the *perlmod* manpage.

Example:

```
sub max {
    my $max = shift(@_);
    foreach $foo (@_) {
        $max = $foo if $max < $foo;
    }
    return $max;
}
$bestday = max($mon,$tue,$wed,$thu,$fri);
```

Example:

```
# get a line, combining continuation lines
#  that start with whitespace

sub get_line {
    $thisline = $lookahead;  # GLOBAL VARIABLES!!
    LINE: while (defined($lookahead = <STDIN>)) {
        if ($lookahead =~ /^[ \t]/) {
            $thisline .= $lookahead;
        }
        else {
            last LINE;
        }
    }
    $thisline;
}

$lookahead = <STDIN>;       # get first line
while ($_ = get_line()) {
    ...
}
```

Use array assignment to a local list to name your formal arguments:

```
sub maybeset {
    my($key, $value) = @_;
    $Foo{$key} = $value unless $Foo{$key};
}
```

This also has the effect of turning call-by-reference into call-by-value, because the assignment copies the values. Otherwise a function is free to do in-place modifications of `@_` and change its caller's values.

```
upcase_in($v1, $v2);  # this changes $v1 and $v2
sub upcase_in {
    for (@_) { tr/a-z/A-Z/ }
}
```

You aren't allowed to modify constants in this way, of course. If an argument were actually literal and you tried to change it, you'd take a (presumably fatal) exception. For example, this won't work:

```
upcase_in("frederick");
```

It would be much safer if the `upcase_in()` function were written to return a copy of its parameters instead of changing them in place:

```
($v3, $v4) = upcase($v1, $v2);  # this doesn't
sub upcase {
    return unless defined wantarray;  # void context, do nothing
    my @parms = @_;
    for (@parms) { tr/a-z/A-Z/ }
    return wantarray ? @parms : $parms[0];
}
```

Notice how this (unprototyped) function doesn't care whether it was passed real scalars or arrays. Perl will see everything as one big long flat `@_` parameter list. This is one of the ways where Perl's simple argument-passing style shines. The `upcase()` function would work perfectly well without changing the `upcase()` definition even if we fed it things like this:

```
@newlist   = upcase(@list1, @list2);
@newlist   = upcase( split /:/, $var );
```

Do not, however, be tempted to do this:

```
(@a, @b)   = upcase(@list1, @list2);
```

Because like its flat incoming parameter list, the return list is also flat. So all you have managed to do here is stored everything in `@a` and made `@b` an empty list. See the section on *Pass by Reference* for alternatives.

A subroutine may be called using the "`&`" prefix. The "`&`" is optional in modern Perls, and so are the parentheses if the subroutine has been predeclared. (Note, however, that the "`&`" is *NOT* optional when you're just naming the subroutine, such as when it's used as an argument to `defined()` or `undef()`. Nor is it optional when you want to do an indirect subroutine call with a subroutine name or reference using the `&$subref()` or `&{$subref}()` constructs. See the *perlref* manpage for more on that.)

Subroutines may be called recursively. If a subroutine is called using the "`&`" form, the argument list is optional, and if omitted, no `@_` array is set up for the subroutine: the `@_` array at the time of the call is visible to subroutine instead. This is an efficiency mechanism that new users may wish to avoid.

```
&foo(1,2,3);        # pass three arguments
foo(1,2,3);         # the same

foo();              # pass a null list
&foo();             # the same

&foo;               # foo() get current args, like foo(@_) !!
foo;                # like foo() IFF sub foo predeclared, else "foo"
```

Not only does the "`&`" form make the argument list optional, but it also disables any prototype checking on the arguments you do provide. This is partly for historical reasons, and partly for having a convenient way to cheat if you know what you're doing. See the section on Prototypes below.

Function whose names are in all upper case are reserved to the Perl core, just as are modules whose names are in all lower case. A function in all capitals is a loosely-held convention meaning it will be called indirectly by the run-time system itself. Functions that do special, pre-defined things are `BEGIN`, `END`, `AUTOLOAD`, and `DESTROY`--plus all the functions mentioned in the *perltie* manpage. The 5.005 release adds `INIT` to this list.

### Private Variables via `my()`

Synopsis:

```
my $foo;            # declare $foo lexically local
my (@wid, %get);    # declare list of variables local
my $foo = "flurp";  # declare $foo lexical, and init it
my @oof = @bar;     # declare @oof lexical, and init it
```

A "`my`" declares the listed variables to be confined (lexically) to the enclosing block, conditional (`if/unless/elsif/else`), loop (`for/foreach/while/until/continue`), subroutine, `eval`, or `do/require/use`'d file. If more than one value is listed, the list must be placed in parentheses. All listed elements must be legal lvalues. Only alphanumeric identifiers may be lexically scoped—magical builtins like `$/` must currently be `localize` with "`local`" instead.

Unlike dynamic variables created by the "`local`" operator, lexical variables declared with "`my`" are totally hidden from the outside world, including any called subroutines (even if it's the same subroutine called from itself or elsewhere—every call gets its own copy).

This doesn't mean that a `my()` variable declared in a statically *enclosing* lexical scope would be invisible. Only the dynamic scopes are cut off. For example, the `bumpx()` function below has access to the lexical `$x` variable because both the my and the sub occurred at the same scope, presumably the file scope.

```
my $x = 10;
sub bumpx { $x++ }
```

(An `eval()`, however, can see the lexical variables of the scope it is being evaluated in so long as the names aren't hidden by declarations within the `eval()` itself. See the *perlref* manpage.)

The parameter list to `my()` may be assigned to if desired, which allows you to initialize your variables. (If no initializer is given for a particular variable, it is created with the undefined value.) Commonly this is used to name the parameters to a subroutine. Examples:

```
    $arg = "fred";        # "global" variable
    $n = cube_root(27);
    print "$arg thinks the root is $n\n";
 fred thinks the root is 3

    sub cube_root {
        my $arg = shift;  # name doesn't matter
        $arg **= 1/3;
        return $arg;
    }
```

The "`my`" is simply a modifier on something you might assign to. So when you do assign to the variables in its argument list, the "`my`" doesn't change whether those variables are viewed as a scalar or an array. So

```
my ($foo) = <STDIN>;                # WRONG?
my @FOO = <STDIN>;
```

both supply a list context to the right-hand side, while

```
my $foo = <STDIN>;
```

supplies a scalar context. But the following declares only one variable:

```
my $foo, $bar = 1;                  # WRONG
```

That has the same effect as

```
my $foo;
$bar = 1;
```

The declared variable is not introduced (is not visible) until after the current statement. Thus,

```
my $x = $x;
```

can be used to initialize the new `$x` with the value of the old `$x`, and the expression

```
my $x = 123 and $x == 123
```

is false unless the old `$x` happened to have the value `123`.

Lexical scopes of control structures are not bounded precisely by the braces that delimit their controlled blocks; control expressions are part of the scope, too. Thus in the loop

```
while (defined(my $line = <>)) {
    $line = lc $line;
} continue {
    print $line;
}
```

the scope of `$line` extends from its declaration throughout the rest of the loop construct (including the `continue` clause), but not beyond it. Similarly, in the conditional

```
if ((my $answer = <STDIN>) =~ /^yes$/i) {
    user_agrees();
} elsif ($answer =~ /^no$/i) {
    user_disagrees();
} else {
    chomp $answer;
    die "'$answer' is neither 'yes' nor 'no'";
}
```

the scope of `$answer` extends from its declaration throughout the rest of the conditional (including `elsif` and `else` clauses, if any), but not beyond it.

(None of the foregoing applies to `if/unless` or `while/until` modifiers appended to simple statements. Such modifiers are not control structures and have no effect on scoping.)

The `foreach` loop defaults to scoping its index variable dynamically (in the manner of `local`; see below). However, if the index variable is prefixed with the keyword "`my`", then it is lexically scoped instead. Thus in the loop

```
for my $i (1, 2, 3) {
    some_function();
}
```

the scope of `$i` extends to the end of the loop, but not beyond it, and so the value of `$i` is unavailable in `some_function()`.

Some users may wish to encourage the use of lexically scoped variables. As an aid to catching implicit references to package variables, if you say

```
use strict 'vars';
```

then any variable reference from there to the end of the enclosing block must either refer to a lexical variable, or must be fully qualified with the package name. A compilation error results otherwise. An inner block may countermand this with "`no strict 'vars'`".

A `my()` has both a compile-time and a run-time effect. At compile time, the compiler takes notice of it;

the principle usefulness of this is to quiet "`use strict 'vars'`". The actual initialization is delayed until run time, so it gets executed appropriately; every time through a loop, for example.

Variables declared with "`my`" are not part of any package and are therefore never fully qualified with the package name. In particular, you're not allowed to try to make a package variable (or other global) lexical:

```
my $pack::var;       # ERROR!  Illegal syntax
my $_;               # also illegal (currently)
```

In fact, a dynamic variable (also known as package or global variables) are still accessible using the fully qualified `::` notation even while a lexical of the same name is also visible:

```
package main;
local $x = 10;
my    $x = 20;
print "$x and $::x\n";
```

That will print out `20` and `10`.

You may declare "`my`" variables at the outermost scope of a file to hide any such identifiers totally from the outside world. This is similar to C's static variables at the file level. To do this with a subroutine requires the use of a closure (anonymous function with lexical access). If a block (such as an `eval()`, function, or `package`) wants to create a private subroutine that cannot be called from outside that block, it can declare a lexical variable containing an anonymous sub reference:

```
my $secret_version = '1.001-beta';
my $secret_sub = sub { print $secret_version };
&$secret_sub();
```

As long as the reference is never returned by any function within the module, no outside module can see the subroutine, because its name is not in any package's symbol table. Remember that it's not *REALLY* called `$some_pack::secret_version` or anything; it's just `$secret_version`, unqualified and unqualifiable.

This does not work with object methods, however; all object methods have to be in the symbol table of some package to be found.

**Peristent Private Variables**

Just because a lexical variable is lexically (also called statically) scoped to its enclosing block, `eval`, or `do` FILE, this doesn't mean that within a function it works like a C static. It normally works more like a C auto, but with implicit garbage collection.

Unlike local variables in C or C++, Perl's lexical variables don't necessarily get recycled just because their scope has exited. If something more permanent is still aware of the lexical, it will stick around. So long as something else references a lexical, that lexical won't be freed—which is as it should be. You wouldn't want memory being free until you were done using it, or kept around once you were done. Automatic garbage collection takes care of this for you.

This means that you can pass back or save away references to lexical variables, whereas to return a pointer to a C auto is a grave error. It also gives us a way to simulate C's function statics. Here's a mechanism for giving a function private variables with both lexical scoping and a static lifetime. If you do want to create something like C's static variables, just enclose the whole function in an extra block, and put the static variable outside the function but in the block.

```
{
    my $secret_val = 0;
    sub gimme_another {
        return ++$secret_val;
    }
}
# $secret_val now becomes unreachable by the outside
# world, but retains its value between calls to gimme_another
```

If this function is being sourced in from a separate file via `require` or `use`, then this is probably just fine. If it's all in the main program, you'll need to arrange for the `my()` to be executed early, either by putting the whole block above your main program, or more likely, placing merely a `BEGIN` sub around it to make sure it gets executed before your program starts to run:

```
sub BEGIN {
    my $secret_val = 0;
    sub gimme_another {
        return ++$secret_val;
    }
}
```

See the section on *Package Constructors and Destructors* in the *perlmod* manpage about the `BEGIN` function.

If declared at the outermost scope, the file scope, then lexicals work someone like C's file statics. They are available to all functions in that same file declared below them, but are inaccessible from outside of the file. This is sometimes used in modules to create private variables for the whole module.

### Temporary Values via *local()*

**NOTE**: In general, you should be using "`my`" instead of "`local`", because it's faster and safer. Exceptions to this include the global punctuation variables, filehandles and formats, and direct manipulation of the Perl symbol table itself. Format variables often use "`local`" though, as do other variables whose current value must be visible to called subroutines.

Synopsis:

```
local $foo;                 # declare $foo dynamically local
local (@wid, %get);         # declare list of variables local
local $foo = "flurp";       # declare $foo dynamic, and init it
local @oof = @bar;          # declare @oof dynamic, and init it

local *FH;                  # localize $FH, @FH, %FH, &FH  ...
local *merlyn = *randal;    # now $merlyn is really $randal, plus
                            #     @merlyn is really @randal, etc
local *merlyn = 'randal';   # SAME THING: promote 'randal' to *randal
local *merlyn = \$randal;   # just alias $merlyn, not @merlyn etc
```

A `local()` modifies its listed variables to be "local" to the enclosing block, `eval`, or `do FILE`--and to *any subroutine called from within that block*. A `local()` just gives temporary values to global (meaning package) variables. It does **not** create a local variable. This is known as dynamic scoping. Lexical scoping is done with "`my`", which works more like C's auto declarations.

If more than one variable is given to `local()`, they must be placed in parentheses. All listed elements must be legal lvalues. This operator works by saving the current values of those variables in its argument list on a hidden stack and restoring them upon exiting the block, subroutine, or eval. This means that called subroutines can also reference the local variable, but not the global one. The argument list may be assigned to if desired, which allows you to initialize your local variables. (If no initializer is given for a particular variable, it is created with an undefined value.) Commonly this is used to name the parameters to a subroutine. Examples:

```
for $i ( 0 .. 9 ) {
    $digits{$i} = $i;
}
# assume this function uses global %digits hash
parse_num();

# now temporarily add to %digits hash
if ($base12) {
    # (NOTE: not claiming this is efficient!)
    local %digits  = (%digits, 't' => 10, 'e' => 11);
    parse_num();  # parse_num gets this new %digits!
}
# old %digits restored here
```

Because `local()` is a run-time command, it gets executed every time through a loop. In releases of Perl previous to 5.0, this used more stack storage each time until the loop was exited. Perl now reclaims the space each time through, but it's still more efficient to declare your variables outside the loop.

A `local` is simply a modifier on an lvalue expression. When you assign to a `local`ized variable, the `local` doesn't change whether its list is viewed as a scalar or an array. So

```
local($foo) = <STDIN>;
local @FOO = <STDIN>;
```

both supply a list context to the right-hand side, while

```
local $foo = <STDIN>;
```

supplies a scalar context.

A note about `local()` and composite types is in order. Something like `local(%foo)` works by temporarily placing a brand new hash in the symbol table. The old hash is left alone, but is hidden "behind" the new one.

This means the old variable is completely invisible via the symbol table (i.e. the hash entry in the `*foo` typeglob) for the duration of the dynamic scope within which the `local()` was seen. This has the effect of allowing one to temporarily occlude any magic on composite types. For instance, this will briefly alter a tied hash to some other implementation:

```
tie %ahash, 'APackage';
[...]
{
   local %ahash;
   tie %ahash, 'BPackage';
   [..called code will see %ahash tied to 'BPackage'..]
   {
      local %ahash;
      [..%ahash is a normal (untied) hash here..]
   }
}
[..%ahash back to its initial tied self again..]
```

As another example, a custom implementation of `%ENV` might look like this:

```
{
    local %ENV;
    tie %ENV, 'MyOwnEnv';
    [..do your own fancy %ENV manipulation here..]
}
[..normal %ENV behavior here..]
```

It's also worth taking a moment to explain what happens when you `local`ize a member of a composite type (i.e. an array or hash element). In this case, the element is `local`ized *by name*. This means that when the scope of the `local()` ends, the saved value will be restored to the hash element whose key was named in the `local()`, or the array element whose index was named in the `local()`. If that element was deleted while the `local()` was in effect (e.g. by a `delete()` from a hash or a `shift()` of an array), it will spring back into existence, possibly extending an array and filling in the skipped elements with `undef`. For instance, if you say

```
%hash = ( 'This' => 'is', 'a' => 'test' );
@ary  = ( 0..5 );
{
     local($ary[5]) = 6;
     local($hash{'a'}) = 'drill';
     while (my $e = pop(@ary)) {
         print "$e . . .\n";
         last unless $e > 3;
     }
     if (@ary) {
         $hash{'only a'} = 'test';
         delete $hash{'a'};
     }
}
print join(' ', map { "$_ $hash{$_}" } sort keys %hash),".\n";
print "The array has ",scalar(@ary)," elements: ",
      join(', ', map { defined $_ ? $_ : 'undef' } @ary),"\n";
```

Perl will print

```
6 . . .
4 . . .
3 . . .
This is a test only a test.
The array has 6 elements: 0, 1, 2, undef, undef, 5
```

**Passing Symbol Table Entries (typeglobs)**

[Note: The mechanism described in this section was originally the only way to simulate pass-by-reference in older versions of Perl. While it still works fine in modern versions, the new reference mechanism is generally easier to work with. See below.]

Sometimes you don't want to pass the value of an array to a subroutine but rather the name of it, so that the subroutine can modify the global copy of it rather than working with a local copy. In perl you can refer to all objects of a particular name by prefixing the name with a star: `*foo`. This is often known as a "typeglob", because the star on the front can be thought of as a wildcard match for all the funny prefix characters on variables and subroutines and such.

When evaluated, the typeglob produces a scalar value that represents all the objects of that name, including any filehandle, format, or subroutine. When assigned to, it causes the name mentioned to refer to whatever "`*`" value was assigned to it. Example:

```
sub doubleary {
    local(*someary) = @_;
    foreach $elem (@someary) {
        $elem *= 2;
    }
}
doubleary(*foo);
doubleary(*bar);
```

Note that scalars are already passed by reference, so you can modify scalar arguments without using this mechanism by referring explicitly to `$_[0]` etc. You can modify all the elements of an array by passing all the elements as scalars, but you have to use the `*` mechanism (or the equivalent reference mechanism) to `push`, `pop`, or change the size of an array. It will certainly be faster to pass the typeglob (or reference).

Even if you don't want to modify an array, this mechanism is useful for passing multiple arrays in a single LIST, because normally the LIST mechanism will merge all the array values so that you can't extract out the individual arrays. For more on typeglobs, see the section on *Typeglobs and Filehandles* in the *perldata* manpage.

### When to Still Use *local()*

Despite the existence of `my()`, there are still three places where the `local()` operator still shines. In fact, in these three places, you *must* use `local` instead of `my`.

1. You need to give a global variable a temporary value, especially `$_`.

   The global variables, like `@ARGV` or the punctuation variables, must be `local`ized with `local()`. This block reads in */etc/motd*, and splits it up into chunks separated by lines of equal signs, which are placed in `@Fields`.

   ```
   {
       local @ARGV = ("/etc/motd");
       local $/ = undef;
       local $_ = <>;
       @Fields = split /^\s*=+\s*$/;
   }
   ```

   It particular, it's important to `localize` `$_` in any routine that assigns to it. Look out for implicit assignments in `while` conditionals.

2. You need to create a local file or directory handle or a local function.

   A function that needs a filehandle of its own must use `local()` uses `local()` on complete typeglob. This can be used to create new symbol table entries:

   ```
   sub ioqueue {
       local  (*READER, *WRITER);    # not my!
       pipe    (READER,  WRITER);    or die "pipe: $!";
       return (*READER, *WRITER);
   }
   ($head, $tail) = ioqueue();
   ```

   See the Symbol module for a way to create anonymous symbol table entries.

   Because assignment of a reference to a typeglob creates an alias, this can be used to create what is effectively a local function, or at least, a local alias.

```
{
    local *grow = \&shrink; # only until this block exists
    grow();                 # really calls shrink()
    move();                 # if move() grow()s, it shrink()s too
}
grow();                     # get the real grow() again
```

See the section on *Function Templates* in the *perlref* manpage for more about manipulating functions by name in this way.

3. You want to temporarily change just one element of an array or hash.

   You can `localize` just one element of an aggregate. Usually this is done on dynamics:

```
{
    local $SIG{INT} = 'IGNORE';
    funct();                            # uninterruptible
}
# interruptibility automatically restored here
```

   But it also works on lexically declared aggregates. Prior to 5.005, this operation could on occasion misbehave.

**Pass by Reference**

If you want to pass more than one array or hash into a function— or return them from it— and have them maintain their integrity, then you're going to have to use an explicit pass-by-reference. Before you do that, you need to understand references as detailed in the *perlref* manpage. This section may not make much sense to you otherwise.

Here are a few simple examples. First, let's pass in several arrays to a function and have it `pop` all of then, return a new list of all their former last elements:

```
@tailings = popmany ( \@a, \@b, \@c, \@d );

sub popmany {
    my $aref;
    my @retlist = ();
    foreach $aref ( @_ ) {
        push @retlist, pop @$aref;
    }
    return @retlist;
}
```

Here's how you might write a function that returns a list of keys occurring in all the hashes passed to it:

```
@common = inter( \%foo, \%bar, \%joe );
sub inter {
    my ($k, $href, %seen); # locals
    foreach $href (@_) {
        while ( $k = each %$href ) {
            $seen{$k}++;
        }
    }
    return grep { $seen{$_} == @_ } keys %seen;
}
```

So far, we're using just the normal list return mechanism. What happens if you want to pass or return a hash? Well, if you're using only one of them, or you don't mind them concatenating, then the normal calling convention is ok, although a little expensive.

Where people get into trouble is here:

```
    (@a, @b) = func(@c, @d);
or
    (%a, %b) = func(%c, %d);
```

That syntax simply won't work. It sets just `@a` or `%a` and clears the `@b` or `%b`. Plus the function didn't get passed into two separate arrays or hashes: it got one long list in `@_`, as always.

If you can arrange for everyone to deal with this through references, it's cleaner code, although not so nice to look at. Here's a function that takes two array references as arguments, returning the two array elements in order of how many elements they have in them:

```
    ($aref, $bref) = func(\@c, \@d);
    print "@$aref has more than @$bref\n";
    sub func {
        my ($cref, $dref) = @_;
        if (@$cref > @$dref) {
            return ($cref, $dref);
        } else {
            return ($dref, $cref);
        }
    }
```

It turns out that you can actually do this also:

```
    (*a, *b) = func(\@c, \@d);
    print "@a has more than @b\n";
    sub func {
        local (*c, *d) = @_;
        if (@c > @d) {
            return (\@c, \@d);
        } else {
            return (\@d, \@c);
        }
    }
```

Here we're using the typeglobs to do symbol table aliasing. It's a tad subtle, though, and also won't work if you're using `my()` variables, because only globals (well, and `local()`s) are in the symbol table.

If you're passing around filehandles, you could usually just use the bare typeglob, like `*STDOUT`, but typeglobs references would be better because they'll still work properly under `use strict 'refs'`. For example:

```
    splutter(\*STDOUT);
    sub splutter {
        my $fh = shift;
        print $fh "her um well a hmmm\n";
    }

    $rec = get_rec(\*STDIN);
    sub get_rec {
        my $fh = shift;
        return scalar <$fh>;
    }
```

Another way to do this is using `*HANDLE{IO}`, see the *perlref* manpage for usage and caveats.

If you're planning on generating new filehandles, you could do this:

```
sub openit {
    my $name = shift;
    local *FH;
    return open (FH, $path) ? *FH : undef;
}
```

Although that will actually produce a small memory leak. See the bottom of the `open()` entry in the *perlfunc* manpage for a somewhat cleaner way using the `IO::Handle` package.

**Prototypes**

As of the 5.002 release of perl, if you declare

```
sub mypush (\@@)
```

then `mypush()` takes arguments exactly like `push()` does. The declaration of the function to be called must be visible at compile time. The prototype affects only the interpretation of new-style calls to the function, where new-style is defined as not using the `&` character. In other words, if you call it like a builtin function, then it behaves like a builtin function. If you call it like an old-fashioned subroutine, then it behaves like an old-fashioned subroutine. It naturally falls out from this rule that prototypes have no influence on subroutine references like `\&foo` or on indirect subroutine calls like `&{$subref}`.

Method calls are not influenced by prototypes either, because the function to be called is indeterminate at compile time, because it depends on inheritance.

Because the intent is primarily to let you define subroutines that work like builtin commands, here are the prototypes for some other functions that parse almost exactly like the corresponding builtins.

```
Declared as                 Called as

sub mylink ($$)          mylink $old, $new
sub myvec ($$$)          myvec $var, $offset, 1
sub myindex ($$;$)       myindex &getstring, "substr"
sub mysyswrite ($$$;$)   mysyswrite $buf, 0, length($buf) - $off, $off
sub myreverse (@)        myreverse $a, $b, $c
sub myjoin ($@)          myjoin ":", $a, $b, $c
sub mypop (\@)           mypop @array
sub mysplice (\@$$@)     mysplice @array, @array, 0, @pushme
sub mykeys (\%)          mykeys %{$hashref}
sub myopen (*;$)         myopen HANDLE, $name
sub mypipe (**)          mypipe READHANDLE, WRITEHANDLE
sub mygrep (&@)          mygrep { /foo/ } $a, $b, $c
sub myrand ($)           myrand 42
sub mytime ()            mytime
```

Any backslashed prototype character represents an actual argument that absolutely must start with that character. The value passed to the subroutine (as part of `@_`) will be a reference to the actual argument given in the subroutine call, obtained by applying \ to that argument.

Unbackslashed prototype characters have special meanings. Any unbackslashed `@` or `%` eats all the rest of the arguments, and forces list context. An argument represented by `$` forces scalar context. An `&` requires an anonymous subroutine, which, if passed as the first argument, does not require the "`sub`" keyword or a subsequent comma. A `*` does whatever it has to do to turn the argument into a reference to a symbol table entry.

A semicolon separates mandatory arguments from optional arguments. (It is redundant before `@` or `%`.)

Note how the last three examples above are treated specially by the parser. `mygrep()` is parsed as a true list operator, `myrand()` is parsed as a true unary operator with unary precedence the same as `rand()`, and `mytime()` is truly without arguments, just like `time()`. That is, if you say

```
mytime +2;
```

you'll get `mytime() + 2`, not `mytime(2)`, which is how it would be parsed without the prototype.

The interesting thing about `&` is that you can generate new syntax with it:

```
sub try (&@) {
    my($try,$catch) = @_;
    eval { &$try };
    if ($@) {
        local $_ = $@;
        &$catch;
    }
}
sub catch (&) { $_[0] }

try {
    die "phooey";
} catch {
    /phooey/ and print "unphooey\n";
};
```

That prints `"unphooey"`. (Yes, there are still unresolved issues having to do with the visibility of `@_`. I'm ignoring that question for the moment. (But note that if we make `@_` lexically scoped, those anonymous subroutines can act like closures... (Gee, is this sounding a little Lispish? (Never mind.))))

And here's a reimplementation of `grep`:

```
sub mygrep (&@) {
    my $code = shift;
    my @result;
    foreach $_ (@_) {
        push(@result, $_) if &$code;
    }
    @result;
}
```

Some folks would prefer full alphanumeric prototypes. Alphanumerics have been intentionally left out of prototypes for the express purpose of someday in the future adding named, formal parameters. The current mechanism's main goal is to let module writers provide better diagnostics for module users. Larry feels the notation quite understandable to Perl programmers, and that it will not intrude greatly upon the meat of the module, nor make it harder to read. The line noise is visually encapsulated into a small pill that's easy to swallow.

It's probably best to prototype new functions, not retrofit prototyping into older ones. That's because you must be especially careful about silent impositions of differing list versus scalar contexts. For example, if you decide that a function should take just one parameter, like this:

```
sub func ($) {
    my $n = shift;
    print "you gave me $n\n";
}
```

and someone has been calling it with an array or expression returning a list:

```
func(@foo);
func( split /:/ );
```

Then you've just supplied an automatic `scalar()` in front of their argument, which can be more than a bit surprising. The old `@foo` which used to hold one thing doesn't get passed in. Instead, the `func()` now gets passed in `1`, that is, the number of elements in `@foo`. And the `split()` gets called in a scalar context and starts scribbling on your `@_` parameter list.

This is all very powerful, of course, and should be used only in moderation to make the world a better place.

**Constant Functions**

Functions with a prototype of `()` are potential candidates for inlining. If the result after optimization and constant folding is either a constant or a lexically-scoped scalar which has no other references, then it will be used in place of function calls made without `&` or `do`. Calls made using `&` or `do` are never inlined. (See *constant.pm* for an easy way to declare most constants.)

The following functions would all be inlined:

```
sub pi ()           { 3.14159 }             # Not exact, but close.
sub PI ()           { 4 * atan2 1, 1 }      # As good as it gets,
                                            # and it's inlined, too!
sub ST_DEV ()       { 0 }
sub ST_INO ()       { 1 }

sub FLAG_FOO ()     { 1 << 8 }
sub FLAG_BAR ()     { 1 << 9 }
sub FLAG_MASK ()    { FLAG_FOO | FLAG_BAR }

sub OPT_BAZ ()      { not (0x1B58 & FLAG_MASK) }
sub BAZ_VAL () {
    if (OPT_BAZ) {
        return 23;
    }
    else {
        return 42;
    }
}

sub N () { int(BAZ_VAL) / 3 }
BEGIN {
    my $prod = 1;
    for (1..N) { $prod *= $_ }
    sub N_FACTORIAL () { $prod }
}
```

If you redefine a subroutine that was eligible for inlining, you'll get a mandatory warning. (You can use this warning to tell whether or not a particular subroutine is considered constant.) The warning is considered severe enough not to be optional because previously compiled invocations of the function will still be using the old value of the function. If you need to be able to redefine the subroutine you need to ensure that it isn't inlined, either by dropping the `()` prototype (which changes the calling semantics, so beware) or by thwarting the inlining mechanism in some other way, such as

```
sub not_inlined () {
    23 if $];
}
```

**Overriding Builtin Functions**

Many builtin functions may be overridden, though this should be tried only occasionally and for good reason. Typically this might be done by a package attempting to emulate missing builtin functionality on a non-Unix system.

Overriding may be done only by importing the name from a module—ordinary predeclaration isn't good enough. However, the `subs` pragma (compiler directive) lets you, in effect, predeclare subs via the import syntax, and these names may then override the builtin ones:

```
use subs 'chdir', 'chroot', 'chmod', 'chown';
chdir $somewhere;
sub chdir { ... }
```

To unambiguously refer to the builtin form, one may precede the builtin name with the special package qualifier `CORE::`. For example, saying `CORE::open()` will always refer to the builtin `open()`, even if the current package has imported some other subroutine called `&open()` from elsewhere.

Library modules should not in general export builtin names like "`open`" or "`chdir`" as part of their default `@EXPORT` list, because these may sneak into someone else's namespace and change the semantics unexpectedly. Instead, if the module adds the name to the `@EXPORT_OK` list, then it's possible for a user to import the name explicitly, but not implicitly. That is, they could say

```
use Module 'open';
```

and it would import the `open` override, but if they said

```
use Module;
```

they would get the default imports without the overrides.

The foregoing mechanism for overriding builtins is restricted, quite deliberately, to the package that requests the import. There is a second method that is sometimes applicable when you wish to override a builtin everywhere, without regard to namespace boundaries. This is achieved by importing a sub into the special namespace `CORE::GLOBAL::`. Here is an example that quite brazenly replaces the `glob` operator with something that understands regular expressions.

```
package REGlob;
require Exporter;
@ISA = 'Exporter';
@EXPORT_OK = 'glob';

sub import {
    my $pkg = shift;
    return unless @_;
    my $sym = shift;
    my $where = ($sym =~ s/^GLOBAL_// ? 'CORE::GLOBAL' : caller(0));
    $pkg->export($where, $sym, @_);
}

sub glob {
    my $pat = shift;
    my @got;
    local(*D);
    if (opendir D, '.') { @got = grep /$pat/, readdir D; closedir D; }
    @got;
}
1;
```

And here's how it could be (ab)used:

```
#use REGlob 'GLOBAL_glob';      # override glob() in ALL namespaces
package Foo;
use REGlob 'glob';              # override glob() in Foo:: only
print for <^[a-z_]+\.pm\$>;     # show all pragmatic modules
```

Note that the initial comment shows a contrived, even dangerous example. By overriding `glob` globally, you would be forcing the new (and subversive) behavior for the `glob` operator for **every** namespace, without the complete cognizance or cooperation of the modules that own those namespaces. Naturally, this should be done with extreme caution— if it must be done at all.

The `REGlob` example above does not implement all the support needed to cleanly override perl's `glob` operator. The builtin `glob` has different behaviors depending on whether it appears in a scalar or list context, but our `REGlob` doesn't. Indeed, many perl builtins have such context sensitive behaviors, and these must be adequately supported by a properly written override. For a fully functional example of overriding `glob`, study the implementation of `File::DosGlob` in the standard library.

### Autoloading

If you call a subroutine that is undefined, you would ordinarily get an immediate fatal error complaining that the subroutine doesn't exist. (Likewise for subroutines being used as methods, when the method doesn't exist in any base class of the class package.) If, however, there is an `AUTOLOAD` subroutine defined in the package or packages that were searched for the original subroutine, then that `AUTOLOAD` subroutine is called with the arguments that would have been passed to the original subroutine. The fully qualified name of the original subroutine magically appears in the `$AUTOLOAD` variable in the same package as the `AUTOLOAD` routine. The name is not passed as an ordinary argument because, er, well, just because, that's why...

Most `AUTOLOAD` routines will load in a definition for the subroutine in question using eval, and then execute that subroutine using a special form of "goto" that erases the stack frame of the `AUTOLOAD` routine without a trace. (See the standard `AutoLoader` module, for example.) But an `AUTOLOAD` routine can also just emulate the routine and never define it. For example, let's pretend that a function that wasn't defined should just call `system()` with those arguments. All you'd do is this:

```
sub AUTOLOAD {
    my $program = $AUTOLOAD;
    $program =~ s/.*:://;
    system($program, @_);
}
date();
who('am', 'i');
ls('-l');
```

In fact, if you predeclare the functions you want to call that way, you don't even need the parentheses:

```
use subs qw(date who ls);
date;
who "am", "i";
ls -l;
```

A more complete example of this is the standard Shell module, which can treat undefined subroutine calls as calls to Unix programs.

Mechanisms are available for modules writers to help split the modules up into autoloadable files. See the standard AutoLoader module described in the *AutoLoader* manpage and in the *AutoSplit* manpage, the standard SelfLoader modules in the *SelfLoader* manpage, and the document on adding C functions to perl code in the *perlxs* manpage.

**SEE ALSO**

See the *perlref* manpage for more about references and closures. See the *perlxs* manpage if you'd like to learn about calling C subroutines from perl. See the *perlmod* manpage to learn about bundling up your functions in separate files.

## NAME

perlmod – Perl modules (packages and symbol tables)

## DESCRIPTION

### Packages

Perl provides a mechanism for alternative namespaces to protect packages from stomping on each other's variables. In fact, there's really no such thing as a global variable in Perl (although some identifiers default to the main package instead of the current one). The package statement declares the compilation unit as being in the given namespace. The scope of the package declaration is from the declaration itself through the end of the enclosing block, `eval`, `sub`, or end of file, whichever comes first (the same scope as the *my()* and *local()* operators). All further unqualified dynamic identifiers will be in this namespace. A package statement only affects dynamic variables—including those you've used *local()* on—but *not* lexical variables created with *my()*. Typically it would be the first declaration in a file to be included by the `require` or `use` operator. You can switch into a package in more than one place; it merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and filehandles in other packages by prefixing the identifier with the package name and a double colon: `$Package::Variable`. If the package name is null, the `main` package is assumed. That is, `$::sail` is equivalent to `$main::sail`.

The old package delimiter was a single quote, but double colon is now the preferred delimiter, in part because it's more readable to humans, and in part because it's more readable to **emacs** macros. It also makes C++ programmers feel like they know what's going on—as opposed to using the single quote as separator, which was there to make Ada programmers feel like they knew what's going on. Because the old-fashioned syntax is still supported for backwards compatibility, if you try to use a string like `"This is $owner's house"`, you'll be accessing `$owner::s`; that is, the `$s` variable in package `owner`, which is probably not what you meant. Use braces to disambiguate, as in `"This is ${owner}'s house"`.

Packages may be nested inside other packages: `$OUTER::INNER::var`. This implies nothing about the order of name lookups, however. All symbols are either local to the current package, or must be fully qualified from the outer package name down. For instance, there is nowhere within package `OUTER` that `$INNER::var` refers to `$OUTER::INNER::var`. It would treat package `INNER` as a totally separate global package.

Only identifiers starting with letters (or underscore) are stored in a package's symbol table. All other symbols are kept in package `main`, including all of the punctuation variables like `$_`. In addition, when unqualified, the identifiers STDIN, STDOUT, STDERR, ARGV, ARGVOUT, ENV, INC, and SIG are forced to be in package `main`, even when used for other purposes than their builtin one. Note also that, if you have a package called `m`, `s`, or `y`, then you can't use the qualified form of an identifier because it will be interpreted instead as a pattern match, a substitution, or a transliteration.

(Variables beginning with underscore used to be forced into package main, but we decided it was more useful for package writers to be able to use leading underscore to indicate private variables and method names. `$_` is still global though.)

*Eval()*ed strings are compiled in the package in which the *eval()* was compiled. (Assignments to `$SIG{}`, however, assume the signal handler specified is in the `main` package. Qualify the signal handler name if you wish to have a signal handler in a package.) For an example, examine *perldb.pl* in the Perl library. It initially switches to the `DB` package so that the debugger doesn't interfere with variables in the script you are trying to debug. At various points, however, it temporarily switches back to the `main` package to evaluate various expressions in the context of the `main` package (or wherever you came from). See the *perldebug* manpage.

The special symbol `__PACKAGE__` contains the current package, but cannot (easily) be used to construct variables.

See the *perlsub* manpage for other scoping issues related to *my()* and *local()*, and the *perlref* manpage regarding closures.

### Symbol Tables

The symbol table for a package happens to be stored in the hash of that name with two colons appended. The main symbol table's name is thus `%main::`, or `%::` for short. Likewise symbol table for the nested package mentioned earlier is named `%OUTER::INNER::`.

The value in each entry of the hash is what you are referring to when you use the `*name` typeglob notation. In fact, the following have the same effect, though the first is more efficient because it does the symbol table lookups at compile time:

```
local *main::foo    = *main::bar;
local $main::{foo}  = $main::{bar};
```

You can use this to print out all the variables in a package, for instance. The standard *dumpvar.pl* library and the CPAN module Devel::Symdump make use of this.

Assignment to a typeglob performs an aliasing operation, i.e.,

```
*dick = *richard;
```

causes variables, subroutines, formats, and file and directory handles accessible via the identifier `richard` also to be accessible via the identifier `dick`. If you want to alias only a particular variable or subroutine, you can assign a reference instead:

```
*dick = \$richard;
```

Which makes `$richard` and `$dick` the same variable, but leaves `@richard` and `@dick` as separate arrays. Tricky, eh?

This mechanism may be used to pass and return cheap references into or from subroutines if you won't want to copy the whole thing. It only works when assigning to dynamic variables, not lexicals.

```
%some_hash = ();                    # can't be my()
*some_hash = fn( \%another_hash );
sub fn {
    local *hashsym = shift;
    # now use %hashsym normally, and you
    # will affect the caller's %another_hash
    my %nhash = (); # do what you want
    return \%nhash;
}
```

On return, the reference will overwrite the hash slot in the symbol table specified by the *some_hash typeglob. This is a somewhat tricky way of passing around references cheaply when you won't want to have to remember to dereference variables explicitly.

Another use of symbol tables is for making "constant" scalars.

```
*PI = \3.14159265358979;
```

Now you cannot alter `$PI`, which is probably a good thing all in all. This isn't the same as a constant subroutine, which is subject to optimization at compile-time. This isn't. A constant subroutine is one prototyped to take no arguments and to return a constant expression. See the *perlsub* manpage for details on these. The `use constant` pragma is a convenient shorthand for these.

You can say `*foo{PACKAGE}` and `*foo{NAME}` to find out what name and package the *foo symbol table entry comes from. This may be useful in a subroutine that gets passed typeglobs as arguments:

```
sub identify_typeglob {
    my $glob = shift;
    print 'You gave me ', *{$glob}{PACKAGE}, '::', *{$glob}{NAME}, "\n";
}
identify_typeglob *foo;
identify_typeglob *bar::baz;
```

This prints

```
You gave me main::foo
You gave me bar::baz
```

The *foo{THING} notation can also be used to obtain references to the individual elements of *foo, see the *perlref* manpage.

### Package Constructors and Destructors

There are two special subroutine definitions that function as package constructors and destructors. These are the `BEGIN` and `END` routines. The `sub` is optional for these routines.

A `BEGIN` subroutine is executed as soon as possible, that is, the moment it is completely defined, even before the rest of the containing file is parsed. You may have multiple `BEGIN` blocks within a file— they will execute in order of definition. Because a `BEGIN` block executes immediately, it can pull in definitions of subroutines and such from other files in time to be visible to the rest of the file. Once a `BEGIN` has run, it is immediately undefined and any code it used is returned to Perl's memory pool. This means you can't ever explicitly call a `BEGIN`.

An `END` subroutine is executed as late as possible, that is, when the interpreter is being exited, even if it is exiting as a result of a *die()* function. (But not if it's polymorphing into another program via `exec`, or being blown out of the water by a signal— you have to trap that yourself (if you can).) You may have multiple `END` blocks within a file— they will execute in reverse order of definition; that is: last in, first out (LIFO).

Inside an `END` subroutine, `$?` contains the value that the script is going to pass to `exit()`. You can modify `$?` to change the exit value of the script. Beware of changing `$?` by accident (e.g. by running something via `system`).

Note that when you use the **–n** and **–p** switches to Perl, `BEGIN` and `END` work just as they do in **awk**, as a degenerate case. As currently implemented (and subject to change, since its inconvenient at best), both `BEGIN` *and* `END` blocks are run when you use the **–c** switch for a compile-only syntax check, although your main code is not.

### Perl Classes

There is no special class syntax in Perl, but a package may function as a class if it provides subroutines to act as methods. Such a package may also derive some of its methods from another class (package) by listing the other package name in its global `@ISA` array (which must be a package global, not a lexical).

For more on this, see the *perltoot* manpage and the *perlobj* manpage.

### Perl Modules

A module is just a package that is defined in a library file of the same name, and is designed to be reusable. It may do this by providing a mechanism for exporting some of its symbols into the symbol table of any package using it. Or it may function as a class definition and make its semantics available implicitly through method calls on the class and its objects, without explicit exportation of any symbols. Or it can do a little of both.

For example, to start a normal module called Some::Module, create a file called Some/Module.pm and start with this template:

```
package Some::Module;  # assumes Some/Module.pm

use strict;

BEGIN {
    use Exporter   ();
    use vars       qw($VERSION @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);

    # set the version for version checking
    $VERSION     = 1.00;
    # if using RCS/CVS, this may be preferred
    $VERSION = do { my @r = (q$Revision: 2.21 $ =~ /\d+/g); sprintf "%d."."%02d"

    @ISA         = qw(Exporter);
    @EXPORT      = qw(&func1 &func2 &func4);
    %EXPORT_TAGS = ( );     # eg: TAG => [ qw!name1 name2! ],

    # your exported package globals go here,
    # as well as any optionally exported functions
    @EXPORT_OK   = qw($Var1 %Hashit &func3);
}
use vars      @EXPORT_OK;

# non-exported package globals go here
use vars      qw(@more $stuff);

# initalize package globals, first exported ones
$Var1   = '';
%Hashit = ();

# then the others (which are still accessible as $Some::Module::stuff)
$stuff  = '';
@more   = ();

# all file-scoped lexicals must be created before
# the functions below that use them.

# file-private lexicals go here
my $priv_var    = '';
my %secret_hash = ();

# here's a file-private function as a closure,
# callable as &$priv_func;  it cannot be prototyped.
my $priv_func = sub {
    # stuff goes here.
};

# make all your functions, whether exported or not;
# remember to put something interesting in the {} stubs
sub func1      {}    # no prototype
sub func2()    {}    # proto'd void
sub func3($$)  {}    # proto'd to 2 scalars
```

```
# this one isn't exported, but could be called!
sub func4(\%)  {}    # proto'd to 1 hash ref

END { }       # module clean-up code here (global destructor)
```

Then go on to declare and use your variables in functions without any qualifications. See the *Exporter* manpage and the the *perlmodlib* manpage for details on mechanics and style issues in module creation.

Perl modules are included into your program by saying

```
use Module;
```

or

```
use Module LIST;
```

This is exactly equivalent to

```
BEGIN { require Module; import Module; }
```

or

```
BEGIN { require Module; import Module LIST; }
```

As a special case

```
use Module ();
```

is exactly equivalent to

```
BEGIN { require Module; }
```

All Perl module files have the extension *.pm*. `use` assumes this so that you don't have to spell out "*Module.pm*" in quotes. This also helps to differentiate new modules from old *.pl* and *.ph* files. Module names are also capitalized unless they're functioning as pragmas, "Pragmas" are in effect compiler directives, and are sometimes called "pragmatic modules" (or even "pragmata" if you're a classicist).

The two statements:

```
require SomeModule;
require "SomeModule.pm";
```

differ from each other in two ways. In the first case, any double colons in the module name, such as `Some::Module`, are translated into your system's directory separator, usually "/". The second case does not, and would have to be specified literally. The other difference is that seeing the first `require` clues in the compiler that uses of indirect object notation involving "SomeModule", as in `$ob = purge SomeModule`, are method calls, not function calls. (Yes, this really can make a difference.)

Because the `use` statement implies a `BEGIN` block, the importation of semantics happens at the moment the `use` statement is compiled, before the rest of the file is compiled. This is how it is able to function as a pragma mechanism, and also how modules are able to declare subroutines that are then visible as list operators for the rest of the current file. This will not work if you use `require` instead of `use`. With require you can get into this problem:

```
require Cwd;                # make Cwd:: accessible
$here = Cwd::getcwd();

use Cwd;                    # import names from Cwd::
$here = getcwd();
```

```
require Cwd;                    # make Cwd:: accessible
$here = getcwd();               # oops! no main::getcwd()
```

In general, `use Module ()` is recommended over `require Module`, because it determines module availability at compile time, not in the middle of your program's execution. An exception would be if two modules each tried to `use` each other, and each also called a function from that other module. In that case, it's easy to use `require`s instead.

Perl packages may be nested inside other package names, so we can have package names containing `::`. But if we used that package name directly as a filename it would makes for unwieldy or impossible filenames on some systems. Therefore, if a module's name is, say, `Text::Soundex`, then its definition is actually found in the library file *Text/Soundex.pm*.

Perl modules always have a *.pm* file, but there may also be dynamically linked executables or autoloaded subroutine definitions associated with the module. If so, these will be entirely transparent to the user of the module. It is the responsibility of the *.pm* file to load (or arrange to autoload) any additional functionality. The POSIX module happens to do both dynamic loading and autoloading, but the user can say just `use POSIX` to get it all.

For more information on writing extension modules, see the *perlxstut* manpage and the *perlguts* manpage.

**SEE ALSO**

See the *perlmodlib* manpage for general style issues related to building Perl modules and classes as well as descriptions of the standard library and CPAN, the *Exporter* manpage for how Perl's standard import/export mechanism works, the *perltoot* manpage for an in-depth tutorial on creating classes, the *perlobj* manpage for a hard-core reference document on objects, and the *perlsub* manpage for an explanation of functions and scoping.

## NAME

perlmodlib − constructing new Perl modules and finding existing ones

## DESCRIPTION

## THE PERL MODULE LIBRARY

A number of modules are included the Perl distribution. These are described below, and all end in *.pm*. You may also discover files in the library directory that end in either *.pl* or *.ph*. These are old libraries supplied so that old programs that use them still run. The *.pl* files will all eventually be converted into standard modules, and the *.ph* files made by **h2ph** will probably end up as extension modules made by **h2xs**. (Some *.ph* values may already be available through the POSIX module.) The **pl2pm** file in the distribution may help in your conversion, but it's just a mechanical process and therefore far from bulletproof.

### Pragmatic Modules

They work somewhat like pragmas in that they tend to affect the compilation of your program, and thus will usually work well only when used within a `use`, or `no`. Most of these are locally scoped, so an inner BLOCK may countermand any of these by saying:

```
no integer;
no strict 'refs';
```

which lasts until the end of that BLOCK.

Unlike the pragmas that effect the `$^H` hints variable, the `use vars` and `use subs` declarations are not BLOCK−scoped. They allow you to predeclare a variables or subroutines within a particular *file* rather than just a block. Such declarations are effective for the entire file for which they were declared. You cannot rescind them with `no vars` or `no subs`.

The following pragmas are defined (and have their own documentation).

use autouse MODULE => qw(sub1 sub2 sub3)
: Defers `require MODULE` until someone calls one of the specified subroutines (which must be exported by MODULE). This pragma should be used with caution, and only when necessary.

| | |
|---|---|
| blib | manipulate `@INC` at compile time to use MakeMaker's uninstalled version of a package |
| diagnostics | force verbose warning diagnostics |
| integer | compute arithmetic in integer instead of double |
| less | request less of something from the compiler |
| lib | manipulate `@INC` at compile time |
| locale | use or ignore current locale for builtin operations (see the *perllocale* manpage) |
| ops | restrict named opcodes when compiling or running Perl code |
| overload | overload basic Perl operations |
| re | alter behaviour of regular expressions |
| sigtrap | enable simple signal handling |
| strict | restrict unsafe constructs |
| subs | predeclare sub names |
| vmsish | adopt certain VMS−specific behaviors |
| vars | predeclare global variable names |

**Standard Modules**

Standard, bundled modules are all expected to behave in a well-defined manner with respect to namespace pollution because they use the Exporter module. See their own documentation for details.

AnyDBM_File
provide framework for multiple DBMs

AutoLoader load functions only on demand

AutoSplit split a package for autoloading

Benchmark benchmark running times of code

CPAN interface to Comprehensive Perl Archive Network

CPAN::FirstTime
create a CPAN configuration file

CPAN::Nox run CPAN while avoiding compiled extensions

Carp warn of errors (from perspective of caller)

Class::Struct declare struct-like datatypes

Config access Perl configuration information

Cwd get pathname of current working directory

DB_File access to Berkeley DB

Devel::SelfStubber
generate stubs for a SelfLoading module

DirHandle supply object methods for directory handles

DynaLoader dynamically load C libraries into Perl code

English use nice English (or awk) names for ugly punctuation variables

Env import environment variables

Exporter implements default import method for modules

ExtUtils::Embed
utilities for embedding Perl in C/C++ applications

ExtUtils::Install
install files from here to there

ExtUtils::Liblist
determine libraries to use and how to use them

ExtUtils::MM_OS2
methods to override Unix behaviour in ExtUtils::MakeMaker

ExtUtils::MM_Unix
methods used by ExtUtils::MakeMaker

ExtUtils::MM_VMS
methods to override Unix behaviour in ExtUtils::MakeMaker

ExtUtils::MakeMaker
create an extension Makefile

ExtUtils::Manifest
utilities to write and check a MANIFEST file

ExtUtils::Mkbootstrap
make a bootstrap file for use by DynaLoader

ExtUtils::Mksymlists
: write linker options files for dynamic extension

ExtUtils::testlib
: add blib/* directories to `@INC`

Fatal
: make errors in builtins or Perl functions fatal

Fcntl
: load the C Fcntl.h defines

File::Basename
: split a pathname into pieces

File::CheckTree
: run many filetest checks on a tree

File::Compare
: compare files or filehandles

File::Copy
: copy files or filehandles

File::Find
: traverse a file tree

File::Path
: create or remove a series of directories

File::stat
: by-name interface to Perl's builtin *stat()* functions

FileCache
: keep more files open than the system permits

FileHandle
: supply object methods for filehandles

FindBin
: locate directory of original Perl script

GDBM_File
: access to the gdbm library

Getopt::Long
: extended processing of command line options

Getopt::Std
: process single-character switches with switch clustering

I18N::Collate
: compare 8-bit scalar data according to the current locale

IO
: load various IO modules

IO::File
: supply object methods for filehandles

IO::Handle
: supply object methods for I/O handles

IO::Pipe
: supply object methods for pipes

IO::Seekable
: supply seek based methods for I/O objects

IO::Select
: OO interface to the select system call

IO::Socket
: object interface to socket communications

IPC::Open2
: open a process for both reading and writing

IPC::Open3
: open a process for reading, writing, and error handling

Math::BigFloat
: arbitrary length float math package

Math::BigInt
: arbitrary size integer math package

Math::Complex
: complex numbers and associated mathematical functions

Math::Trig
: simple interface to parts of Math::Complex for those who need trigonometric functions only for real numbers

NDBM_File
: tied access to ndbm files

| | |
|---|---|
| Net::Ping | Hello, anybody home? |
| Net::hostent | by-name interface to Perl's builtin gethost*() functions |
| Net::netent | by-name interface to Perl's builtin getnet*() functions |
| Net::protoent | by-name interface to Perl's builtin getproto*() functions |
| Net::servent | by-name interface to Perl's builtin getserv*() functions |
| Opcode | disable named opcodes when compiling or running Perl code |
| Pod::Text | convert POD data to formatted ASCII text |
| POSIX | interface to IEEE Standard 1003.1 |
| SDBM_File | tied access to sdbm files |
| Safe | compile and execute code in restricted compartments |
| Search::Dict | search for key in dictionary file |
| SelectSaver | save and restore selected file handle |
| SelfLoader | load functions only on demand |
| Shell | run shell commands transparently within Perl |
| Socket | load the C socket.h defines and structure manipulators |
| Symbol | manipulate Perl symbols and their names |
| Sys::Hostname | try every conceivable way to get hostname |
| Sys::Syslog | interface to the Unix *syslog* (3) calls |
| Term::Cap | termcap interface |
| Term::Complete | word completion module |
| Term::ReadLine | interface to various `readline` packages |
| Test::Harness | run Perl standard test scripts with statistics |
| Text::Abbrev | create an abbreviation table from a list |
| Text::ParseWords | parse text into an array of tokens |
| Text::Soundex | implementation of the Soundex Algorithm as described by Knuth |
| Text::Tabs | expand and unexpand tabs per the Unix *expand* (1) and *unexpand* (1) |
| Text::Wrap | line wrapping to form simple paragraphs |
| Tie::Hash | base class definitions for tied hashes |
| Tie::RefHash | base class definitions for tied hashes with references as keys |
| Tie::Scalar | base class definitions for tied scalars |
| Tie::SubstrHash | fixed-table-size, fixed-key-length hashing |
| Time::Local | efficiently compute time from local and GMT time |
| Time::gmtime | by-name interface to Perl's builtin *gmtime()* function |

| | |
|---|---|
| Time::localtime | by-name interface to Perl's builtin *localtime()* function |
| Time::tm | internal object used by Time::gmtime and Time::localtime |
| UNIVERSAL | base class for ALL classes (blessed references) |
| User::grent | by-name interface to Perl's builtin getgr*() functions |
| User::pwent | by-name interface to Perl's builtin getpw*() functions |

To find out *all* the modules installed on your system, including those without documentation or outside the standard release, do this:

```
% find `perl -e 'print "@INC"'` -name '*.pm' -print
```

They should all have their own documentation installed and accessible via your system *man* (1) command. If that fails, try the *perldoc* program.

### Extension Modules

Extension modules are written in C (or a mix of Perl and C) and may be statically linked or in general are dynamically loaded into Perl if and when you need them. Supported extension modules include the Socket, Fcntl, and POSIX modules.

Many popular C extension modules do not come bundled (at least, not completely) due to their sizes, volatility, or simply lack of time for adequate testing and configuration across the multitude of platforms on which Perl was beta-tested. You are encouraged to look for them in *archie* (1L), the Perl FAQ or Meta-FAQ, the WWW page, and even with their authors before randomly posting asking for their present condition and disposition.

## CPAN

CPAN stands for the Comprehensive Perl Archive Network. This is a globally replicated collection of all known Perl materials, including hundreds of unbundled modules. Here are the major categories of modules:

- Language Extensions and Documentation Tools
- Development Support
- Operating System Interfaces
- Networking, Device Control (modems) and InterProcess Communication
- Data Types and Data Type Utilities
- Database Interfaces
- User Interfaces
- Interfaces to / Emulations of Other Programming Languages
- File Names, File Systems and File Locking (see also File Handles)
- String Processing, Language Text Processing, Parsing, and Searching
- Option, Argument, Parameter, and Configuration File Processing
- Internationalization and Locale
- Authentication, Security, and Encryption
- World Wide Web, HTML, HTTP, CGI, MIME
- Server and Daemon Utilities
- Archiving and Compression

- Images, Pixmap and Bitmap Manipulation, Drawing, and Graphing
- Mail and Usenet News
- Control Flow Utilities (callbacks and exceptions etc)
- File Handle and Input/Output Stream Utilities
- Miscellaneous Modules

The registered CPAN sites as of this writing include the following. You should try to choose one close to you:

- Africa

```
South Africa    ftp://ftp.is.co.za/programming/perl/CPAN/
```

- Asia

```
Hong Kong       ftp://ftp.hkstar.com/pub/CPAN/
Japan           ftp://ftp.jaist.ac.jp/pub/lang/perl/CPAN/
                ftp://ftp.lab.kdd.co.jp/lang/perl/CPAN/
South Korea     ftp://ftp.nuri.net/pub/CPAN/
Taiwan          ftp://dongpo.math.ncu.edu.tw/perl/CPAN/
                ftp://ftp.wownet.net/pub2/PERL/
```

- Australasia

```
Australia       ftp://ftp.netinfo.com.au/pub/perl/CPAN/
New Zealand     ftp://ftp.tekotago.ac.nz/pub/perl/CPAN/
```

- Europe

```
Austria         ftp://ftp.tuwien.ac.at/pub/languages/perl/CPAN/
Belgium         ftp://ftp.kulnet.kuleuven.ac.be/pub/mirror/CPAN/
Czech Republic  ftp://sunsite.mff.cuni.cz/Languages/Perl/CPAN/
Denmark         ftp://sunsite.auc.dk/pub/languages/perl/CPAN/
Finland         ftp://ftp.funet.fi/pub/languages/perl/CPAN/
France          ftp://ftp.ibp.fr/pub/perl/CPAN/
                ftp://ftp.pasteur.fr/pub/computing/unix/perl/CPAN/
Germany         ftp://ftp.gmd.de/packages/CPAN/
                ftp://ftp.leo.org/pub/comp/programming/languages/perl/CPAN/
                ftp://ftp.mpi-sb.mpg.de/pub/perl/CPAN/
                ftp://ftp.rz.ruhr-uni-bochum.de/pub/CPAN/
                ftp://ftp.uni-erlangen.de/pub/source/Perl/CPAN/
                ftp://ftp.uni-hamburg.de/pub/soft/lang/perl/CPAN/
Greece          ftp://ftp.ntua.gr/pub/lang/perl/
Hungary         ftp://ftp.kfki.hu/pub/packages/perl/CPAN/
Italy           ftp://cis.utovrm.it/CPAN/
the Netherlands ftp://ftp.cs.ruu.nl/pub/PERL/CPAN/
                ftp://ftp.EU.net/packages/cpan/
Norway          ftp://ftp.uit.no/pub/languages/perl/cpan/
Poland          ftp://ftp.pk.edu.pl/pub/lang/perl/CPAN/
                ftp://sunsite.icm.edu.pl/pub/CPAN/
Portugal        ftp://ftp.ci.uminho.pt/pub/lang/perl/
                ftp://ftp.telepac.pt/pub/CPAN/
Russia          ftp://ftp.sai.msu.su/pub/lang/perl/CPAN/
Slovenia        ftp://ftp.arnes.si/software/perl/CPAN/
Spain           ftp://ftp.etse.urv.es/pub/mirror/perl/
                ftp://ftp.rediris.es/mirror/CPAN/
Sweden          ftp://ftp.sunet.se/pub/lang/perl/CPAN/
UK              ftp://ftp.demon.co.uk/pub/mirrors/perl/CPAN/
                ftp://sunsite.doc.ic.ac.uk/packages/CPAN/
                ftp://unix.hensa.ac.uk/mirrors/perl-CPAN/
```

• North America

```
Ontario         ftp://ftp.utilis.com/public/CPAN/
                ftp://enterprise.ic.gc.ca/pub/perl/CPAN/
Manitoba        ftp://theory.uwinnipeg.ca/pub/CPAN/
California      ftp://ftp.digital.com/pub/plan/perl/CPAN/
                ftp://ftp.cdrom.com/pub/perl/CPAN/
Colorado        ftp://ftp.cs.colorado.edu/pub/perl/CPAN/
Florida         ftp://ftp.cis.ufl.edu/pub/perl/CPAN/
Illinois        ftp://uiarchive.uiuc.edu/pub/lang/perl/CPAN/
Massachusetts   ftp://ftp.iguide.com/pub/mirrors/packages/perl/CPAN/
New York        ftp://ftp.rge.com/pub/languages/perl/
North Carolina  ftp://ftp.duke.edu/pub/perl/
Oklahoma        ftp://ftp.ou.edu/mirrors/CPAN/
Oregon          http://www.perl.org/CPAN/
                ftp://ftp.orst.edu/pub/packages/CPAN/
Pennsylvania    ftp://ftp.epix.net/pub/languages/perl/
Texas           ftp://ftp.sedl.org/pub/mirrors/CPAN/
                ftp://ftp.metronet.com/pub/perl/
```

- South America

```
Chile          ftp://sunsite.dcc.uchile.cl/pub/Lang/perl/CPAN/
```

For an up-to-date listing of CPAN sites, see *http://www.perl.com/perl/CPAN* or *ftp://ftp.perl.com/perl/*.

**Modules: Creation, Use, and Abuse**

(The following section is borrowed directly from Tim Bunce's modules file, available at your nearest CPAN site.)

Perl implements a class using a package, but the presence of a package doesn't imply the presence of a class. A package is just a namespace. A class is a package that provides subroutines that can be used as methods. A method is just a subroutine that expects, as its first argument, either the name of a package (for "static" methods), or a reference to something (for "virtual" methods).

A module is a file that (by convention) provides a class of the same name (sans the .pm), plus an import method in that class that can be called to fetch exported symbols. This module may implement some of its methods by loading dynamic C or C++ objects, but that should be totally transparent to the user of the module. Likewise, the module might set up an AUTOLOAD function to slurp in subroutine definitions on demand, but this is also transparent. Only the *.pm* file is required to exist. See the *perlsub* manpage, the *perltoot* manpage, and the *AutoLoader* manpage for details about the AUTOLOAD mechanism.

### Guidelines for Module Creation

Do similar modules already exist in some form?

If so, please try to reuse the existing modules either in whole or by inheriting useful features into a new class. If this is not practical try to get together with the module authors to work on extending or enhancing the functionality of the existing modules. A perfect example is the plethora of packages in perl4 for dealing with command line options.

If you are writing a module to expand an already existing set of modules, please coordinate with the author of the package. It helps if you follow the same naming scheme and module interaction scheme as the original author.

Try to design the new module to be easy to extend and reuse.

Use blessed references. Use the two argument form of bless to bless into the class name given as the first parameter of the constructor, e.g.,:

```
sub new {
       my $class = shift;
       return bless {}, $class;
}
```

or even this if you'd like it to be used as either a static or a virtual method.

```
sub new {
       my $self  = shift;
       my $class = ref($self) || $self;
       return bless {}, $class;
}
```

Pass arrays as references so more parameters can be added later (it's also faster). Convert functions into methods where appropriate. Split large methods into smaller more flexible ones. Inherit methods from other modules if appropriate.

Avoid class name tests like: `die "Invalid" unless ref $ref eq 'FOO'`. Generally you can delete the "`eq 'FOO'`" part with no harm at all. Let the objects look after themselves! Generally, avoid hard-wired class names as far as possible.

Avoid `$r->Class::func()` where using `@ISA=qw(... Class ...)` and `$r->func()`

would work (see the *perlbot* manpage for more details).

Use autosplit so little used or newly added functions won't be a burden to programs that don't use them. Add test functions to the module after __END__ either using AutoSplit or by saying:

```
eval join('',<main::DATA>) || die $@ unless caller();
```

Does your module pass the 'empty subclass' test? If you say "`@SUBCLASS::ISA = qw(YOURCLASS);`" your applications should be able to use SUBCLASS in exactly the same way as YOURCLASS. For example, does your application still work if you change: `$obj = new YOURCLASS;` into: `$obj = new SUBCLASS;` ?

Avoid keeping any state information in your packages. It makes it difficult for multiple other packages to use yours. Keep state information in objects.

Always use **-w**. Try to `use strict;` (or `use strict qw(...);`). Remember that you can add `no strict qw(...);` to individual blocks of code that need less strictness. Always use **-w**. Always use **-w**! Follow the guidelines in the *perlstyle* (1) manual.

Some simple style guidelines

The perlstyle manual supplied with Perl has many helpful points.

Coding style is a matter of personal taste. Many people evolve their style over several years as they learn what helps them write and maintain good code. Here's one set of assorted suggestions that seem to be widely used by experienced developers:

Use underscores to separate words. It is generally easier to read `$var_names_like_this` than `$VarNamesLikeThis`, especially for non-native speakers of English. It's also a simple rule that works consistently with VAR_NAMES_LIKE_THIS.

Package/Module names are an exception to this rule. Perl informally reserves lowercase module names for 'pragma' modules like integer and strict. Other modules normally begin with a capital letter and use mixed case with no underscores (need to be short and portable).

You may find it helpful to use letter case to indicate the scope or nature of a variable. For example:

```
$ALL_CAPS_HERE   constants only (beware clashes with Perl vars)
$Some_Caps_Here  package-wide global/static
$no_caps_here    function scope my() or local() variables
```

Function and method names seem to work best as all lowercase. e.g., `$obj->as_string()`.

You can use a leading underscore to indicate that a variable or function should not be used outside the package that defined it.

Select what to export.

Do NOT export method names!

Do NOT export anything else by default without a good reason!

Exports pollute the namespace of the module user. If you must export try to use `@EXPORT_OK` in preference to `@EXPORT` and avoid short or common names to reduce the risk of name clashes.

Generally anything not exported is still accessible from outside the module using the ModuleName::item_name (or `$blessed_ref->method`) syntax. By convention you can use a leading underscore on names to indicate informally that they are 'internal' and not for public use.

(It is actually possible to get private functions by saying: `my $subref = sub { ... }; &$subref;`. But there's no way to call that directly as a method, because a method must have a name in the symbol table.)

As a general rule, if the module is trying to be object oriented then export nothing. If it's just a collection of functions then `@EXPORT_OK` anything but use `@EXPORT` with caution.

Select a name for the module.

This name should be as descriptive, accurate, and complete as possible. Avoid any risk of ambiguity. Always try to use two or more whole words. Generally the name should reflect what is special about what the module does rather than how it does it. Please use nested module names to group informally or categorize a module. There should be a very good reason for a module not to have a nested name. Module names should begin with a capital letter.

Having 57 modules all called Sort will not make life easy for anyone (though having 23 called Sort::Quick is only marginally better :−). Imagine someone trying to install your module alongside many others. If in any doubt ask for suggestions in comp.lang.perl.misc.

If you are developing a suite of related modules/classes it's good practice to use nested classes with a common prefix as this will avoid namespace clashes. For example: Xyz::Control, Xyz::View, Xyz::Model etc. Use the modules in this list as a naming guide.

If adding a new module to a set, follow the original author's standards for naming modules and the interface to methods in those modules.

To be portable each component of a module name should be limited to 11 characters. If it might be used on MS−DOS then try to ensure each is unique in the first 8 characters. Nested modules make this easier.

Have you got it right?

How do you know that you've made the right decisions? Have you picked an interface design that will cause problems later? Have you picked the most appropriate name? Do you have any questions?

The best way to know for sure, and pick up many helpful suggestions, is to ask someone who knows. Comp.lang.perl.misc is read by just about all the people who develop modules and it's the best place to ask.

All you need to do is post a short summary of the module, its purpose and interfaces. A few lines on each of the main methods is probably enough. (If you post the whole module it might be ignored by busy people − generally the very people you want to read it!)

Don't worry about posting if you can't say when the module will be ready − just say so in the message. It might be worth inviting others to help you, they may be able to complete it for you!

README and other Additional Files.

It's well known that software developers usually fully document the software they write. If, however, the world is in urgent need of your software and there is not enough time to write the full documentation please at least provide a README file containing:

- A description of the module/package/extension etc.
- A copyright notice − see below.
- Prerequisites − what else you may need to have.
- How to build it − possible changes to Makefile.PL etc.
- How to install it.
- Recent changes in this release, especially incompatibilities
- Changes / enhancements you plan to make in the future.

If the README file seems to be getting too large you may wish to split out some of the sections into separate files: INSTALL, Copying, ToDo etc.

Adding a Copyright Notice.

How you choose to license your work is a personal decision. The general mechanism is to assert your Copyright and then make a declaration of how others may copy/use/modify your work.

Perl, for example, is supplied with two types of licence: The GNU GPL and The Artistic Licence (see the files README, Copying, and Artistic). Larry has good reasons for NOT just using the

GNU GPL.

My personal recommendation, out of respect for Larry, Perl, and the Perl community at large is to state something simply like:

```
Copyright (c) 1995 Your Name. All rights reserved.
This program is free software; you can redistribute it and/or
modify it under the same terms as Perl itself.
```

This statement should at least appear in the README file. You may also wish to include it in a Copying file and your source files. Remember to include the other words in addition to the Copyright.

Give the module a version/issue/release number.

To be fully compatible with the Exporter and MakeMaker modules you should store your module's version number in a non-my package variable called `$VERSION`. This should be a floating point number with at least two digits after the decimal (i.e., hundredths, e.g, `$VERSION = "0.01"`). Don't use a "1.3.2" style version. See Exporter.pm in Perl5.001m or later for details.

It may be handy to add a function or method to retrieve the number. Use the number in announcements and archive file names when releasing the module (ModuleName-1.02.tar.Z). See perldoc ExtUtils::MakeMaker.pm for details.

How to release and distribute a module.

It's good idea to post an announcement of the availability of your module (or the module itself if small) to the comp.lang.perl.announce Usenet newsgroup. This will at least ensure very wide once-off distribution.

If possible you should place the module into a major ftp archive and include details of its location in your announcement.

Some notes about ftp archives: Please use a long descriptive file name that includes the version number. Most incoming directories will not be readable/listable, i.e., you won't be able to see your file after uploading it. Remember to send your email notification message as soon as possible after uploading else your file may get deleted automatically. Allow time for the file to be processed and/or check the file has been processed before announcing its location.

FTP Archives for Perl Modules:

Follow the instructions and links on

```
http://franz.ww.tu-berlin.de/modulelist
```

or upload to one of these sites:

```
ftp://franz.ww.tu-berlin.de/incoming
ftp://ftp.cis.ufl.edu/incoming
```

and notify <*upload@franz.ww.tu-berlin.de*>.

By using the WWW interface you can ask the Upload Server to mirror your modules from your ftp or WWW site into your own directory on CPAN!

Please remember to send me an updated entry for the Module list!

Take care when changing a released module.

Always strive to remain compatible with previous released versions. Otherwise try to add a mechanism to revert to the old behaviour if people rely on it. Document incompatible changes.

### Guidelines for Converting Perl 4 Library Scripts into Modules

There is no requirement to convert anything.
If it ain't broke, don't fix it! Perl 4 library scripts should continue to work with no problems. You may need to make some minor changes (like escaping non-array @'s in double quoted strings) but there is no need to convert a .pl file into a Module for just that.

Consider the implications.
All Perl applications that make use of the script will need to be changed (slightly) if the script is converted into a module. Is it worth it unless you plan to make other changes at the same time?

Make the most of the opportunity.
If you are going to convert the script to a module you can use the opportunity to redesign the interface. The 'Guidelines for Module Creation' above include many of the issues you should consider.

The pl2pm utility will get you started.
This utility will read *.pl files (given as parameters) and write corresponding *.pm files. The pl2pm utilities does the following:

- Adds the standard Module prologue lines
- Converts package specifiers from ' to ::
- Converts die(...) to croak(...)
- Several other minor changes

Being a mechanical process pl2pm is not bullet proof. The converted code will need careful checking, especially any package statements. Don't delete the original .pl file till the new .pm one works!

### Guidelines for Reusing Application Code

Complete applications rarely belong in the Perl Module Library.

Many applications contain some Perl code that could be reused.
Help save the world! Share your code in a form that makes it easy to reuse.

Break-out the reusable code into one or more separate module files.

Take the opportunity to reconsider and redesign the interfaces.

In some cases the 'application' can then be reduced to a small
fragment of code built on top of the reusable modules. In these cases the application could invoked as:

```
     % perl -e 'use Module::Name; method(@ARGV)' ...
or
     % perl -mModule::Name ...    (in perl5.002 or higher)
```

## NOTE

Perl does not enforce private and public parts of its modules as you may have been used to in other languages like C++, Ada, or Modula-17. Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun.

The module and its user have a contract, part of which is common law, and part of which is "written". Part of the common law contract is that a module doesn't pollute any namespace it wasn't asked to. The written contract for the module (A.K.A. documentation) may make other provisions. But then you know when you `use RedefineTheWorld` that you're redefining the world and willing to take the consequences.

## NAME

perlmodinstall − Installing CPAN Modules

## DESCRIPTION

You can think of a module as the fundamental unit of reusable Perl code; see the *perlmod* manpage for details. Whenever anyone creates a chunk of Perl code that they think will be useful to the world, they register as a Perl developer at http://www.perl.com/CPAN/modules/04pause.html so that they can then upload their code to the CPAN. The CPAN is the Comprehensive Perl Archive Network and can be accessed at http://www.perl.com/CPAN/.

This documentation is for people who want to download CPAN modules and install them on their own computer.

### PREAMBLE

You have a file ending in .tar.gz (or, less often, .zip). You know there's a tasty module inside. There are four steps you must now take:

**DECOMPRESS** the file

**UNPACK** the file into a directory

**BUILD** the module (sometimes unnecessary)

**INSTALL** the module.

Here's how to perform each step for each operating system. This is *not* a substitute for reading the README and INSTALL files that might have come with your module!

Also note that these instructions are tailored for installing the module into your system's repository of Perl modules. But you can install modules into any directory you wish. For instance, where I say `perl Makefile.PL`, you can substitute `perl Makefile.PL PREFIX=/my/perl_directory` to install the modules into `/my/perl_directory`. Then you can use the modules from your Perl programs with `use lib "/my/perl_directory/lib/site_perl";` or sometimes just `use "/my/perl_directory";`.

- **If you're on Unix,**

  You can use Andreas Koenig's CPAN module ( http://www.perl.com/CPAN/modules/by-module/CPAN ) to automate the following steps, from DECOMPRESS through INSTALL.

  A. DECOMPRESS

  Decompress the file with `gzip -d yourmodule.tar.gz`

  You can get gzip from ftp://prep.ai.mit.edu/pub/gnu.

  Or, you can combine this step with the next to save disk space:

  ```
  gzip -dc yourmodule.tar.gz | tar -xof -
  ```

  B. UNPACK

  Unpack the result with `tar -xof yourmodule.tar`

  C. BUILD

  Go into the newly-created directory and type:

  ```
  perl Makefile.PL
  make
  make test
  ```

  D. INSTALL

  While still in that directory, type:

```
make install
```

Make sure you have the appropriate permissions to install the module in your Perl 5 library directory. Often, you'll need to be root.

That's all you need to do on Unix systems with dynamic linking. Most Unix systems have dynamic linking -- if yours doesn't, or if for another reason you have a statically-linked perl, **and** the module requires compilation, you'll need to build a new Perl binary that includes the module. Again, you'll probably need to be root.

- **If you're running Windows 95 or NT with the ActiveState port of Perl**

```
A. DECOMPRESS
```

You can use the shareware Winzip ( http://www.winzip.com ) to decompress and unpack modules.

```
B. UNPACK
```

If you used WinZip, this was already done for you.

```
C. BUILD
```

Does the module require compilation (i.e. does it have files that end in .xs, .c, .h, .y, .cc, .cxx, or .C)? If it does, you're on your own. You can try compiling it yourself if you have a C compiler. If you're successful, consider uploading the resulting binary to the CPAN for others to use. If it doesn't, go to INSTALL.

```
D. INSTALL
```

Copy the module into your Perl's *lib* directory. That'll be one of the directories you see when you type

```
perl -e 'print "@INC"'
```

- **If you're running Windows 95 or NT with the core Windows distribution of Perl,**

```
A. DECOMPRESS
```

When you download the module, make sure it ends in either `.tar.gz` or `.zip`. Windows browsers sometimes download `.tar.gz` files as `_tar.tar`, because early versions of Windows prohibited more than one dot in a filename.

You can use the shareware WinZip ( http://www.winzip.com ) to decompress and unpack modules.

Or, you can use InfoZip's `unzip` utility ( http://www.cdrom.com/pub/infozip/Info-Zip.html ) to uncompress `.zip` files; type `unzip yourmodule.zip` in your shell.

Or, if you have a working `tar` and `gzip`, you can type

```
gzip -cd yourmodule.tar.gz | tar xvf -
```

in the shell to decompress `yourmodule.tar.gz`. This will UNPACK your module as well.

```
B. UNPACK
```

All of the methods in DECOMPRESS will have done this for you.

```
C. BUILD
```

Go into the newly-created directory and type:

```
perl Makefile.PL
dmake
dmake test
```

Depending on your perl configuration, `dmake` might not be available. You might have to substitute whatever `perl -V:make` says. (Usually, that will be `nmake` or `make`.)

```
D. INSTALL
```

While still in that directory, type:

```
dmake install
```

- **If you're using a Macintosh,**

  A. DECOMPRESS

  You can either use StuffIt Expander ( http://www.aladdinsys.com/ ) in combination with *DropStuff with Expander Enhancer* (shareware), or the freeware MacGzip ( http://persephone.cps.unizar.es/general/gente/spd/gzip/gzip.html ).

  B. UNPACK

  If you're using DropStuff or Stuffit, you can just extract the tar archive. Otherwise, you can use the freeware *suntar* ( http://www.cirfid.unibo.it/~speranza ).

  C. BUILD

  Does the module require compilation?

  1. If it does,

  Overview: You need MPW and a combination of new and old CodeWarrior compilers for MPW and libraries. Makefiles created for building under MPW use the Metrowerks compilers. It's most likely possible to build without other compilers, but it has not been done successfully, to our knowledge. Read the documentation in MacPerl: Power and Ease ( http://www.ptf.com/macperl/ ) on porting/building extensions, or find an existing precompiled binary, or hire someone to build it for you.

  Or, ask someone on the mac-perl mailing list (mac-perl@iis.ee.ethz.ch) to build it for you. To subscribe to the mac-perl mailing list, send mail to mac-perl-request@iis.ee.ethz.ch.

  2. If the module doesn't require compilation, go to INSTALL.

  D. INSTALL

  Make sure the newlines for the modules are in Mac format, not Unix format. Move the files manually into the correct folders.

  Move the files to their final destination: This will most likely be in `$ENV{MACPERL}site_lib:` (i.e., `HD:MacPerl folder:site_lib:`). You can add new paths to the default `@INC` in the Preferences menu item in the MacPerl application (`$ENV{MACPERL}site_lib:` is added automagically). Create whatever directory structures are required (i.e., for `Some::Module`, create `$ENV{MACPERL}site_lib:Some:` and put `Module.pm` in that directory).

  Run the following script (or something like it):

```
#!perl -w
use AutoSplit;
my $dir = "${MACPERL}site_perl";
autosplit("$dir:Some:Module.pm", "$dir:auto", 0, 1, 1);
```

  Eventually there should be a way to automate the installation process; some solutions exist, but none are ready for the general public yet.

- **If you're on the DJGPP port of DOS,**

```
A. DECOMPRESS
```

djtarx ( ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2/ ) will both uncompress and unpack.

```
B. UNPACK
```

See above.

```
C. BUILD
```

Go into the newly-created directory and type:

```
perl Makefile.PL
make
make test
```

You will need the packages mentioned in `Readme.dos` in the Perl distribution.

```
D. INSTALL
```

While still in that directory, type:

```
make install
```

```
You will need the packages mentioned in Readme.dos in the Perl distribution.
```

- **If you're on OS/2,**

Get the EMX development suite and gzip/tar, from either Hobbes ( http://hobbes.nmsu.edu ) or Leo ( http://www.leo.org ), and then follow the instructions for Unix.

- **If you're on VMS,**

When downloading from CPAN, save your file with a `.tgz` extension instead of `.tar.gz`. All other periods in the filename should be replaced with underscores. For example, `Your-Module-1.33.tar.gz` should be downloaded as `Your-Module-1_33.tgz`.

A. DECOMPRESS

Type

```
gzip -d Your-Module.tgz
```

or, for zipped modules, type

```
unzip Your-Module.zip
```

Executables for gzip, zip, and VMStar ( Alphas: http://www.openvms.digital.com/cd/000TOOLS/ALPHA/ and Vaxen: http://www.openvms.digital.com/cd/000TOOLS/VAX/ ).

gzip and tar are also available at ftp://ftp.digital.com/pub/VMS.

Note that GNU's gzip/gunzip is not the same as Info-ZIP's zip/unzip package. The former is a simple compression tool; the latter permits creation of multi-file archives.

B. UNPACK

If you're using VMStar:

```
VMStar xf Your-Module.tar
```

Or, if you're fond of VMS command syntax:

```
tar/extract/verbose Your_Module.tar
```

C. BUILD

Make sure you have MMS (from Digital) or the freeware MMK ( available from MadGoat at http://www.madgoat.com ). Then type this to create the DESCRIP.MMS for the module:

```
perl Makefile.PL
```

Now you're ready to build:

```
mms
mms test
```

Substitute `mmk` for `mms` above if you're using MMK.

D. INSTALL

Type

```
mms install
```

Substitute `mmk` for `mms` above if you're using MMK.

- **If you're on MVS**,

  Introduce the .tar.gz file into an HFS as binary; don't translate from ASCII to EBCDIC.

  A. DECOMPRESS

  ```
  Decompress the file with C<gzip -d yourmodule.tar.gz>

  You can get gzip from
  http://www.s390.ibm.com/products/oe/bpxqp1.html.
  ```

  B. UNPACK

  Unpack the result with

  ```
  pax -o to=IBM-1047,from=ISO8859-1 -r < yourmodule.tar
  ```

  The BUILD and INSTALL steps are identical to those for Unix. Some modules generate Makefiles that work better with GNU make, which is available from http://www.mks.com/s390/gnu/index.htm.

**HEY**

If you have any suggested changes for this page, let me know. Please don't send me mail asking for help on how to install your modules. There are too many modules, and too few Orwants, for me to be able to answer or even acknowledge all your questions. Contact the module author instead, or post to comp.lang.perl.modules, or ask someone familiar with Perl on your operating system.

**AUTHOR**

Jon Orwant

orwant@tpj.com

The Perl Journal, http://tpj.com

with invaluable help from Brandon Allbery, Charles Bailey, Graham Barr, Dominic Dunlop, Jarkko Hietaniemi, Ben Holzman, Tom Horsley, Nick Ing-Simmons, Tuomas J. Lukka, Laszlo Molnar, Chris Nandor, Alan Olsen, Peter Prymmer, Gurusamy Sarathy, Christoph Spalinger, Dan Sugalski, Larry Virden, and Ilya Zakharevich.

July 22, 1998

**COPYRIGHT**

**NAME**

perlform − Perl formats

**DESCRIPTION**

Perl has a mechanism to help you generate simple reports and charts. To facilitate this, Perl helps you code up your output page close to how it will look when it's printed. It can keep track of things like how many lines are on a page, what page you're on, when to print page headers, etc. Keywords are borrowed from FORTRAN: *format()* to declare and *write()* to execute; see their entries in the *perlfunc* manpage. Fortunately, the layout is much more legible, more like BASIC's PRINT USING statement. Think of it as a poor man's *nroff*(1).

Formats, like packages and subroutines, are declared rather than executed, so they may occur at any point in your program. (Usually it's best to keep them all together though.) They have their own namespace apart from all the other ''types'' in Perl. This means that if you have a function named ''Foo'', it is not the same thing as having a format named ''Foo''. However, the default name for the format associated with a given filehandle is the same as the name of the filehandle. Thus, the default format for STDOUT is named ''STDOUT'', and the default format for filehandle TEMP is named ''TEMP''. They just look the same. They aren't.

Output record formats are declared as follows:

```
format NAME =
FORMLIST
.
```

If name is omitted, format ''STDOUT'' is defined. FORMLIST consists of a sequence of lines, each of which may be one of three types:

1. A comment, indicated by putting a '#' in the first column.
2. A ''picture'' line giving the format for one output line.
3. An argument line supplying values to plug into the previous picture line.

Picture lines are printed exactly as they look, except for certain fields that substitute values into the line. Each field in a picture line starts with either ''@'' (at) or ''^'' (caret). These lines do not undergo any kind of variable interpolation. The at field (not to be confused with the array marker @) is the normal kind of field; the other kind, caret fields, are used to do rudimentary multi-line text block filling. The length of the field is supplied by padding out the field with multiple ''<'', ''>'', or ''|'' characters to specify, respectively, left justification, right justification, or centering. If the variable would exceed the width specified, it is truncated.

As an alternate form of right justification, you may also use ''#'' characters (with an optional ''.") to specify a numeric field. This way you can line up the decimal points. If any value supplied for these fields contains a newline, only the text up to the newline is printed. Finally, the special field ''@*'' can be used for printing multi-line, nontruncated values; it should appear by itself on a line.

The values are specified on the following line in the same order as the picture fields. The expressions providing the values should be separated by commas. The expressions are all evaluated in a list context before the line is processed, so a single list expression could produce multiple list elements. The expressions may be spread out to more than one line if enclosed in braces. If so, the opening brace must be the first token on the first line. If an expression evaluates to a number with a decimal part, and if the corresponding picture specifies that the decimal part should appear in the output (that is, any picture except multiple ''#'' characters **without** an embedded ''."), the character used for the decimal point is **always** determined by the current LC_NUMERIC locale. This means that, if, for example, the run-time environment happens to specify a German locale, '','' will be used instead of the default ''.''. See the *perllocale* manpage and the section on *WARNINGS* for more information.

Picture fields that begin with ^ rather than @ are treated specially. With a # field, the field is blanked out if the value is undefined. For other field types, the caret enables a kind of fill mode. Instead of an arbitrary expression, the value supplied must be a scalar variable name that contains a text string. Perl puts as much text as it can into the field, and then chops off the front of the string so that the next time the variable is referenced, more of the text can be printed. (Yes, this means that the variable itself is altered during execution of the *write()* call, and is not returned.) Normally you would use a sequence of fields in a vertical stack to print out a block of text. You might wish to end the final field with the text "...", which will appear in the output if the text was too long to appear in its entirety. You can change which characters are legal to break on by changing the variable `$:` (that's `$FORMAT_LINE_BREAK_CHARACTERS` if you're using the English module) to a list of the desired characters.

Using caret fields can produce variable length records. If the text to be formatted is short, you can suppress blank lines by putting a "~" (tilde) character anywhere in the line. The tilde will be translated to a space upon output. If you put a second tilde contiguous to the first, the line will be repeated until all the fields on the line are exhausted. (If you use a field of the at variety, the expression you supply had better not give the same value every time forever!)

Top-of-form processing is by default handled by a format with the same name as the current filehandle with "_TOP" concatenated to it. It's triggered at the top of each page. See the `write` entry in the *perlfunc* manpage.

Examples:

```
# a report on the /etc/passwd file
format STDOUT_TOP =
                        Passwd File
Name                Login    Office   Uid   Gid Home
------------------------------------------------------------------
.
format STDOUT =
@<<<<<<<<<<<<<<<<<< @||||||| @<<<<<<@>>>> @>>>> @<<<<<<<<<<<<<<<<<
$name,              $login,  $office,$uid,$gid, $home
.
```

```
# a report from a bug report form
format STDOUT_TOP =
                        Bug Reports
@<<<<<<<<<<<<<<<<<<<<<<<     @|||         @>>>>>>>>>>>>>>>>>>>>>>>
$system,                      $%,         $date
------------------------------------------------------------------
.
format STDOUT =
Subject: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
         $subject
Index: @<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
       $index,                       $description
Priority: @<<<<<<<<<< Date: @<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
          $priority,        $date,   $description
From: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      $from,                         $description
Assigned to: @<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
             $programmer,            $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<...
                                     $description
.
```

It is possible to intermix *print()*s with *write()*s on the same output channel, but you'll have to handle `$-` (`$FORMAT_LINES_LEFT`) yourself.

### Format Variables

The current format name is stored in the variable `$~` (`$FORMAT_NAME`), and the current top of form format name is in `$^` (`$FORMAT_TOP_NAME`). The current output page number is stored in `$%` (`$FORMAT_PAGE_NUMBER`), and the number of lines on the page is in `$=` (`$FORMAT_LINES_PER_PAGE`). Whether to autoflush output on this handle is stored in `$|` (`$OUTPUT_AUTOFLUSH`). The string output before each top of page (except the first) is stored in `$^L` (`$FORMAT_FORMFEED`). These variables are set on a per-filehandle basis, so you'll need to *select()* into a different one to affect them:

```
    select((select(OUTF),
            $~ = "My_Other_Format",
            $^ = "My_Top_Format"
           )[0]);
```

Pretty ugly, eh? It's a common idiom though, so don't be too surprised when you see it. You can at least use a temporary variable to hold the previous filehandle: (this is a much better approach in general, because not only does legibility improve, you now have intermediary stage in the expression to single-step the debugger through):

```
$ofh = select(OUTF);
$~ = "My_Other_Format";
$^ = "My_Top_Format";
select($ofh);
```

If you use the English module, you can even read the variable names:

```
use English;
$ofh = select(OUTF);
$FORMAT_NAME     = "My_Other_Format";
$FORMAT_TOP_NAME = "My_Top_Format";
select($ofh);
```

But you still have those funny *select()*s. So just use the FileHandle module. Now, you can access these special variables using lowercase method names instead:

```
use FileHandle;
format_name     OUTF "My_Other_Format";
format_top_name OUTF "My_Top_Format";
```

Much better!

**NOTES**

Because the values line may contain arbitrary expressions (for at fields, not caret fields), you can farm out more sophisticated processing to other functions, like *sprintf()* or one of your own. For example:

```
format Ident =
    @<<<<<<<<<<<<<<<
    &commify($n)
.
```

To get a real at or caret into the field, do this:

```
format Ident =
I have an @ here.
        "@"
.
```

To center a whole line of text, do something like this:

```
format Ident =
@|||||||||||||||||||||||||||||||||||||||||||||||
        "Some text line"
.
```

There is no builtin way to say “float this to the right hand side of the page, however wide it is.” You have to specify where it goes. The truly desperate can generate their own format on the fly, based on the current number of columns, and then *eval()* it:

```
$format  = "format STDOUT = \n"
         . '^' . '<' x $cols . "\n"
         . '$entry' . "\n"
         . "\t^" . "<" x ($cols-8) . "~~\n"
         . '$entry' . "\n"
         . ".\n";
print $format if $Debugging;
eval $format;
die $@ if $@;
```

Which would generate a format looking something like this:

```
format STDOUT =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$entry
        ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<~~
$entry
.
```

Here's a little program that's somewhat like *fmt* (1):

```
format =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ~~
$_

.

$/ = '';
while (<>) {
    s/\s*\n\s*/ /g;
    write;
}
```

### Footers

While `$FORMAT_TOP_NAME` contains the name of the current header format, there is no corresponding mechanism to automatically do the same thing for a footer. Not knowing how big a format is going to be until you evaluate it is one of the major problems. It's on the TODO list.

Here's one strategy: If you have a fixed-size footer, you can get footers by checking `$FORMAT_LINES_LEFT` before each *write()* and print the footer yourself if necessary.

Here's another strategy: Open a pipe to yourself, using `open(MYSELF, "|-")` (see the `open()` entry in the *perlfunc* manpage) and always *write()* to MYSELF instead of STDOUT. Have your child process massage its STDIN to rearrange headers and footers however you like. Not very convenient, but doable.

### Accessing Formatting Internals

For low-level access to the formatting mechanism. you may use *formline()* and access `$^A` (the `$ACCUMULATOR` variable) directly.

For example:

```
$str = formline <<'END', 1,2,3;
@<<<  @|||  @>>>
END

print "Wow, I just stored `$^A' in the accumulator!\n";
```

Or to make an *swrite()* subroutine, which is to *write()* what *sprintf()* is to *printf()*, do this:

```
    use Carp;
    sub swrite {
        croak "usage: swrite PICTURE ARGS" unless @_;
        my $format = shift;
        $^A = "";
        formline($format,@_);
        return $^A;
    }

    $string = swrite(<<'END', 1, 2, 3);
 Check me out
 @<<<  @|||  @>>>
 END
    print $string;
```

**WARNINGS**

The lone dot that ends a format can also prematurely end a mail message passing through a misconfigured Internet mailer (and based on experience, such misconfiguration is the rule, not the exception). So when sending format code through mail, you should indent it so that the format-ending dot is not on the left margin; this will prevent SMTP cutoff.

Lexical variables (declared with "my") are not visible within a format unless the format is declared within the scope of the lexical variable. (They weren't visible at all before version 5.001.)

Formats are the only part of Perl that unconditionally use information from a program's locale; if a program's environment specifies an LC_NUMERIC locale, it is always used to specify the decimal point character in formatted output. Perl ignores all other aspects of locale handling unless the `use locale` pragma is in effect. Formatted output cannot be controlled by `use locale` because the pragma is tied to the block structure of the program, and, for historical reasons, formats exist outside that block structure. See the *perllocale* manpage for further discussion of locale handling.

## NAME

perllocale − Perl locale handling (internationalization and localization)

## DESCRIPTION

Perl supports language-specific notions of data such as "is this a letter", "what is the uppercase equivalent of this letter", and "which of these letters comes first". These are important issues, especially for languages other than English—but also for English: it would be naïve to imagine that `A-Za-z` defines all the "letters" needed to write in English. Perl is also aware that some character other than '.' may be preferred as a decimal point, and that output date representations may be language-specific. The process of making an application take account of its users' preferences in such matters is called **internationalization** (often abbreviated as **i18n**); telling such an application about a particular set of preferences is known as **localization** (**l10n**).

Perl can understand language-specific data via the standardized (ISO C, XPG4, POSIX 1.c) method called "the locale system". The locale system is controlled per application using one pragma, one function call, and several environment variables.

**NOTE**: This feature is new in Perl 5.004, and does not apply unless an application specifically requests it—see the section on *Backward compatibility*. The one exception is that *write()* now **always** uses the current locale − see the section on *NOTES*.

## PREPARING TO USE LOCALES

If Perl applications are to understand and present your data correctly according a locale of your choice, **all** of the following must be true:

- **Your operating system must support the locale system**. If it does, you should find that the *setlocale()* function is a documented part of its C library.
- **Definitions for locales that you use must be installed**. You, or your system administrator, must make sure that this is the case. The available locales, the location in which they are kept, and the manner in which they are installed all vary from system to system. Some systems provide only a few, hard-wired locales and do not allow more to be added. Others allow you to add "canned" locales provided by the system supplier. Still others allow you or the system administrator to define and add arbitrary locales. (You may have to ask your supplier to provide canned locales that are not delivered with your operating system.) Read your system documentation for further illumination.
- **Perl must believe that the locale system is supported**. If it does, `perl -V:d_setlocale` will say that the value for `d_setlocale` is `define`.

If you want a Perl application to process and present your data according to a particular locale, the application code should include the `use locale` pragma (see the section on *The use locale pragma*) where appropriate, and **at least one** of the following must be true:

- **The locale-determining environment variables (see the section on** *ENVIRONMENT*) must be correctly set up at the time the application is started, either by yourself or by whoever set up your system account.
- **The application must set its own locale** using the method described in the section on *The setlocale function*.

## USING LOCALES

### The use locale pragma

By default, Perl ignores the current locale. The `use locale` pragma tells Perl to use the current locale for some operations:

- **The comparison operators** (`lt`, `le`, `cmp`, `ge`, and `gt`) and the POSIX string collation functions *strcoll()* and *strxfrm()* use `LC_COLLATE`. *sort()* is also affected if used without an explicit comparison function, because it uses `cmp` by default.

  **Note:** `eq` and `ne` are unaffected by locale: they always perform a byte-by-byte comparison of their scalar operands. What's more, if `cmp` finds that its operands are equal according to the collation sequence specified by the current locale, it goes on to perform a byte-by-byte comparison, and only returns *0* (equal) if the operands are bit-for-bit identical. If you really want to know whether two strings— which `eq` and `cmp` may consider different— are equal as far as collation in the locale is concerned, see the discussion in the section on *Category LC_COLLATE: Collation*.

- **Regular expressions and case-modification functions** (*uc()*, *lc()*, *ucfirst()*, and *lcfirst()*) use `LC_CTYPE`
- **The formatting functions** (*printf()*, *sprintf()* and *write()*) use `LC_NUMERIC`
- **The POSIX date formatting function** (*strftime()*) uses `LC_TIME`.

`LC_COLLATE`, `LC_CTYPE`, and so on, are discussed further in the section on *LOCALE CATEGORIES*.

The default behavior is restored with the `no locale` pragma, or upon reaching the end of block enclosing `use locale`.

The string result of any operation that uses locale information is tainted, as it is possible for a locale to be untrustworthy. See the section on *SECURITY*.

### The setlocale function

You can switch locales as often as you wish at run time with the *POSIX::setlocale()* function:

```
# This functionality not usable prior to Perl 5.004
require 5.004;

# Import locale-handling tool set from POSIX module.
# This example uses: setlocale -- the function call
#                    LC_CTYPE -- explained below
use POSIX qw(locale_h);

# query and save the old locale
$old_locale = setlocale(LC_CTYPE);

setlocale(LC_CTYPE, "fr_CA.ISO8859-1");
# LC_CTYPE now in locale "French, Canada, codeset ISO 8859-1"

setlocale(LC_CTYPE, "");
# LC_CTYPE now reset to default defined by LC_ALL/LC_CTYPE/LANG
# environment variables.  See below for documentation.

# restore the old locale
setlocale(LC_CTYPE, $old_locale);
```

The first argument of *setlocale()* gives the **category**, the second the **locale**. The category tells in what aspect of data processing you want to apply locale-specific rules. Category names are discussed in the section on *LOCALE CATEGORIES* and the section on *ENVIRONMENT*. The locale is the name of a collection of customization information corresponding to a particular combination of language, country or territory, and codeset. Read on for hints on the naming of locales: not all systems name locales as in the example.

If no second argument is provided and the category is something else than LC_ALL, the function returns a string naming the current locale for the category. You can use this value as the second argument in a subsequent call to *setlocale()*.

If no second argument is provided and the category is LC_ALL, the result is implementation-dependent. It may be a string of concatenated locales names (separator also implementation-dependent) or a single locale name. Please consult your the *setlocale(3)* manpage for details.

If a second argument is given and it corresponds to a valid locale, the locale for the category is set to that value, and the function returns the now-current locale value. You can then use this in yet another call to *setlocale()*. (In some implementations, the return value may sometimes differ from the value you gave as the second argument— think of it as an alias for the value you gave.)

As the example shows, if the second argument is an empty string, the category's locale is returned to the default specified by the corresponding environment variables. Generally, this results in a return to the default that was in force when Perl started up: changes to the environment made by the application after startup may or may not be noticed, depending on your system's C library.

If the second argument does not correspond to a valid locale, the locale for the category is not changed, and the function returns *undef*.

For further information about the categories, consult the *setlocale(3)* manpage.

**Finding locales**

For locales available in your system, consult also the *setlocale(3)* manpage to see whether it leads to the list of available locales (search for the *SEE ALSO* section). If that fails, try the following command lines:

```
locale -a

nlsinfo

ls /usr/lib/nls/loc

ls /usr/lib/locale

ls /usr/lib/nls
```

and see whether they list something resembling these

```
en_US.ISO8859-1     de_DE.ISO8859-1     ru_RU.ISO8859-5
en_US.iso88591      de_DE.iso88591      ru_RU.iso88595
en_US               de_DE               ru_RU
en                  de                  ru
english             german              russian
english.iso88591    german.iso88591     russian.iso88595
english.roman8                          russian.koi8r
```

Sadly, even though the calling interface for *setlocale()* has been standardized, names of locales and the directories where the configuration resides have not been. The basic form of the name is *language_country/territory***.***codeset*, but the latter parts after *language* are not always present. The *language* and *country* are usually from the standards **ISO 3166** and **ISO 639**, the two-letter abbreviations for the countries and the languages of the world, respectively. The *codeset* part often mentions some **ISO 8859** character set, the Latin codesets. For example, `ISO 8859-1` is the so-called ''Western codeset'' that can be used to encode most Western European languages. Again, there are several ways to write even the name of that one standard. Lamentably.

Two special locales are worth particular mention: ''C'' and ''POSIX''. Currently these are effectively the same locale: the difference is mainly that the first one is defined by the C standard, the second by the POSIX standard. They define the **default locale** in which every program starts in the absence of locale information in its environment. (The *default* default locale, if you will.) Its language is (American) English and its character codeset ASCII.

**NOTE**: Not all systems have the ''POSIX'' locale (not all systems are POSIX−conformant), so use ''C'' when you need explicitly to specify this default locale.

**LOCALE PROBLEMS**

You may encounter the following warning message at Perl startup:

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
        LC_ALL = "En_US",
        LANG = (unset)
    are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

This means that your locale settings had LC_ALL set to "En_US" and LANG exists but has no value. Perl tried to believe you but could not. Instead, Perl gave up and fell back to the "C" locale, the default locale that is supposed to work no matter what. This usually means your locale settings were wrong, they mention locales your system has never heard of, or the locale installation in your system has problems (for example, some system files are broken or missing). There are quick and temporary fixes to these problems, as well as more thorough and lasting fixes.

**Temporarily fixing locale problems**

The two quickest fixes are either to render Perl silent about any locale inconsistencies or to run Perl under the default locale "C".

Perl's moaning about locale problems can be silenced by setting the environment variable PERL_BADLANG to a non-zero value, for example "1". This method really just sweeps the problem under the carpet: you tell Perl to shut up even when Perl sees that something is wrong. Do not be surprised if later something locale-dependent misbehaves.

Perl can be run under the "C" locale by setting the environment variable LC_ALL to "C". This method is perhaps a bit more civilized than the PERL_BADLANG approach, but setting LC_ALL (or other locale variables) may affect other programs as well, not just Perl. In particular, external programs run from within Perl will see these changes. If you make the new settings permanent (read on), all programs you run see the changes. See the *ENVIRONMENT* manpage for for the full list of relevant environment variables and the section on *USING LOCALES* for their effects in Perl. Effects in other programs are easily deducible. For example, the variable LC_COLLATE may well affect your **sort** program (or whatever the program that arranges 'records' alphabetically in your system is called).

You can test out changing these variables temporarily, and if the new settings seem to help, put those settings into your shell startup files. Consult your local documentation for the exact details. For in Bourne-like shells (**sh**, **ksh**, **bash**, **zsh**):

```
LC_ALL=en_US.ISO8859-1
export LC_ALL
```

This assumes that we saw the locale "en_US.ISO8859-1" using the commands discussed above. We decided to try that instead of the above faulty locale "En_US"— and in Cshish shells (**csh**, **tcsh**)

```
setenv LC_ALL en_US.ISO8859-1
```

```
If you do not know what shell you have, consult your local
helpdesk or the equivalent.
```

**Permanently fixing locale problems**

The slower but superior fixes are when you may be able to yourself fix the misconfiguration of your own environment variables. The *mis*(sing)configuration of the whole system's locales usually requires the help of your friendly system administrator.

First, see earlier in this document about the section on *Finding locales*. That tells how to find which locales are really supported— and more importantly, installed— on your system. In our example error message, environment variables affecting the locale are listed in the order of decreasing importance (and unset variables do not matter). Therefore, having LC_ALL set to "En_US" must have been the bad choice, as shown by the error message. First try fixing locale settings listed first.

Second, if using the listed commands you see something **exactly** (prefix matches do not count and case usually counts) like "En_US" without the quotes, then you should be okay because you are using a locale name that should be installed and available in your system. In this case, see the section on *Fixing system locale configuration*.

### Permanently fixing your locale configuration

This is when you see something like:

```
perl: warning: Please check that your locale settings:
        LC_ALL = "En_US",
        LANG = (unset)
    are supported and installed on your system.
```

but then cannot see that "En_US" listed by the above-mentioned commands. You may see things like "en_US.ISO8859-1", but that isn't the same. In this case, try running under a locale that you can list and which somehow matches what you tried. The rules for matching locale names are a bit vague because standardization is weak in this area. See again the the section on *Finding locales* about general rules.

### Permanently fixing system locale configuration

Contact a system administrator (preferably your own) and report the exact error message you get, and ask them to read this same documentation you are now reading. They should be able to check whether there is something wrong with the locale configuration of the system. The the section on *Finding locales* section is unfortunately a bit vague about the exact commands and places because these things are not that standardized.

### The localeconv function

The *POSIX::localeconv()* function allows you to get particulars of the locale-dependent numeric formatting information specified by the current `LC_NUMERIC` and `LC_MONETARY` locales. (If you just want the name of the current locale for a particular category, use *POSIX::setlocale()* with a single parameter—see the section on *The setlocale function*.)

```
use POSIX qw(locale_h);

# Get a reference to a hash of locale-dependent info
$locale_values = localeconv();

# Output sorted list of the values
for (sort keys %$locale_values) {
    printf "%-20s = %s\n", $_, $locale_values->{$_}
}
```

*localeconv()* takes no arguments, and returns **a reference to** a hash. The keys of this hash are variable names for formatting, such as `decimal_point` and `thousands_sep`. The values are the corresponding, er, values. See the `localeconv` entry in the *POSIX (3)* manpage for a longer example listing the categories an implementation might be expected to provide; some provide more and others fewer. You don't need an explicit `use locale`, because *localeconv()* always observes the current locale.

Here's a simple-minded example program that rewrites its command-line parameters as integers correctly formatted in the current locale:

```
# See comments in previous example
require 5.004;
use POSIX qw(locale_h);

# Get some of locale's numeric formatting parameters
my ($thousands_sep, $grouping) =
     @{localeconv()}{'thousands_sep', 'grouping'};

# Apply defaults if values are missing
$thousands_sep = ',' unless $thousands_sep;

# grouping and mon_grouping are packed lists
# of small integers (characters) telling the
# grouping (thousand_seps and mon_thousand_seps
# being the group dividers) of numbers and
# monetary quantities.  The integers' meanings:
# 255 means no more grouping, 0 means repeat
# the previous grouping, 1-254 means use that
# as the current grouping.  Grouping goes from
# right to left (low to high digits).  In the
# below we cheat slightly by never using anything
# else than the first grouping (whatever that is).
if ($grouping) {
    @grouping = unpack("C*", $grouping);
} else {
    @grouping = (3);
}

# Format command line params for current locale
for (@ARGV) {
    $_ = int;    # Chop non-integer part
    1 while
    s/(\d)(\d{$grouping[0]}($|$thousands_sep))/$1$thousands_sep$2/;
    print "$_";
}
print "\n";
```

## LOCALE CATEGORIES

The following subsections describe basic locale categories. Beyond these, some combination categories allow manipulation of more than one basic category at a time. See the section on *ENVIRONMENT* for a discussion of these.

### Category LC_COLLATE: Collation

In the scope of `use locale`, Perl looks to the `LC_COLLATE` environment variable to determine the application's notions on collation (ordering) of characters. For example, 'b' follows 'a' in Latin alphabets, but where do 'á' and 'å' belong? And while 'color' follows 'chocolate' in English, what about in Spanish?

The following collations all make sense and you may meet any of them if you "use locale".

```
A B C D E a b c d e
A a B b C c D d D e
a A b B c C d D e E
a b c d e A B C D E
```

Here is a code snippet to tell what alphanumeric characters are in the current locale, in that locale's order:

```
use locale;
print +(sort grep /\w/, map { chr() } 0..255), "\n";
```

Compare this with the characters that you see and their order if you state explicitly that the locale should be ignored:

```
no locale;
print +(sort grep /\w/, map { chr() } 0..255), "\n";
```

This machine-native collation (which is what you get unless `use locale` has appeared earlier in the same block) must be used for sorting raw binary data, whereas the locale-dependent collation of the first example is useful for natural text.

As noted in the section on *USING LOCALES*, `cmp` compares according to the current collation locale when `use locale` is in effect, but falls back to a byte-by-byte comparison for strings that the locale says are equal. You can use *POSIX::strcoll()* if you don't want this fall-back:

```
use POSIX qw(strcoll);
$equal_in_locale =
    !strcoll("space and case ignored", "SpaceAndCaseIgnored");
```

$equal_in_locale will be true if the collation locale specifies a dictionary-like ordering that ignores space characters completely and which folds case.

If you have a single string that you want to check for ‘‘equality in locale’’ against several others, you might think you could gain a little efficiency by using *POSIX::strxfrm()* in conjunction with `eq`:

```
use POSIX qw(strxfrm);
$xfrm_string = strxfrm("Mixed-case string");
print "locale collation ignores spaces\n"
    if $xfrm_string eq strxfrm("Mixed-casestring");
print "locale collation ignores hyphens\n"
    if $xfrm_string eq strxfrm("Mixedcase string");
print "locale collation ignores case\n"
    if $xfrm_string eq strxfrm("mixed-case string");
```

*strxfrm()* takes a string and maps it into a transformed string for use in byte-by-byte comparisons against other transformed strings during collation. ‘‘Under the hood’’, locale-affected Perl comparison operators call *strxfrm()* for both operands, then do a byte-by-byte comparison of the transformed strings. By calling *strxfrm()* explicitly and using a non locale-affected comparison, the example attempts to save a couple of transformations. But in fact, it doesn't save anything: Perl magic (see the section on *Magic Variables* in the *perlguts* manpage) creates the transformed version of a string the first time it's needed in a comparison, then keeps this version around in case it's needed again. An example rewritten the easy way with `cmp` runs just about as fast. It also copes with null characters embedded in strings; if you call *strxfrm()* directly, it treats the first null it finds as a terminator. don't expect the transformed strings it produces to be portable across systems—or even from one revision of your operating system to the next. In short, don't call *strxfrm()* directly: let Perl do it for you.

Note: `use locale` isn't shown in some of these examples because it isn't needed: *strcoll()* and *strxfrm()* exist only to generate locale-dependent results, and so always obey the current `LC_COLLATE` locale.

### Category LC_CTYPE: Character Types

In the scope of `use locale`, Perl obeys the `LC_CTYPE` locale setting. This controls the application's notion of which characters are alphabetic. This affects Perl's `\w` regular expression metanotation, which stands for alphanumeric characters—that is, alphabetic and numeric characters. (Consult the *perlre* manpage for more information about regular expressions.) Thanks to `LC_CTYPE`, depending on your locale setting, characters like ‘æ’, ‘∂’, ‘β’, and ’ø’ may be understood as `\w` characters.

The `LC_CTYPE` locale also provides the map used in transliterating characters between lower and uppercase. This affects the case-mapping functions--*lc()*, lcfirst, *uc()*, and *ucfirst()*; case-mapping interpolation with `\l`, `\L`, `\u`, or `\U` in double-quoted strings and `s///` substitutions; and case-independent regular expression pattern matching using the `i` modifier.

Finally, `LC_CTYPE` affects the POSIX character-class test functions--*isalpha()*, *islower()*, and so on. For example, if you move from the "C" locale to a 7-bit Scandinavian one, you may find— possibly to your surprise— that "|" moves from the *ispunct()* class to *isalpha()*.

**Note:** A broken or malicious `LC_CTYPE` locale definition may result in clearly ineligible characters being considered to be alphanumeric by your application. For strict matching of (mundane) letters and digits— for example, in command strings— locale-aware applications should use `\w` inside a `no locale` block. See the section on *SECURITY*.

### Category LC_NUMERIC: Numeric Formatting

In the scope of `use locale`, Perl obeys the `LC_NUMERIC` locale information, which controls an application's idea of how numbers should be formatted for human readability by the *printf()*, *sprintf()*, and *write()* functions. String-to-numeric conversion by the *POSIX::strtod()* function is also affected. In most implementations the only effect is to change the character used for the decimal point— perhaps from '.' to ','. These functions aren't aware of such niceties as thousands separation and so on. (See the section on *The localeconv function* if you care about these things.)

Output produced by *print()* is **never** affected by the current locale: it is independent of whether `use locale` or `no locale` is in effect, and corresponds to what you'd get from *printf()* in the "C" locale. The same is true for Perl's internal conversions between numeric and string formats:

```
use POSIX qw(strtod);
use locale;

$n = 5/2;   # Assign numeric 2.5 to $n

$a = " $n"; # Locale-independent conversion to string

print "half five is $n\n";       # Locale-independent output

printf "half five is %g\n", $n;  # Locale-dependent output

print "DECIMAL POINT IS COMMA\n"
    if $n == (strtod("2,5"))[0]; # Locale-dependent conversion
```

### Category LC_MONETARY: Formatting of monetary amounts

The C standard defines the `LC_MONETARY` category, but no function that is affected by its contents. (Those with experience of standards committees will recognize that the working group decided to punt on the issue.) Consequently, Perl takes no notice of it. If you really want to use `LC_MONETARY`, you can query its contents— see the section on *The localeconv function*--and use the information that it returns in your application's own formatting of currency amounts. However, you may well find that the information, voluminous and complex though it may be, still does not quite meet your requirements: currency formatting is a hard nut to crack.

### LC_TIME

Output produced by *POSIX::strftime()*, which builds a formatted human-readable date/time string, is affected by the current `LC_TIME` locale. Thus, in a French locale, the output produced by the `%B` format element (full month name) for the first month of the year would be "janvier". Here's how to get a list of long month names in the current locale:

```
use POSIX qw(strftime);
for (0..11) {
    $long_month_name[$_] =
        strftime("%B", 0, 0, 0, 1, $_, 96);
}
```

Note: `use locale` isn't needed in this example: as a function that exists only to generate locale-dependent results, *strftime()* always obeys the current `LC_TIME` locale.

### Other categories

The remaining locale category, `LC_MESSAGES` (possibly supplemented by others in particular implementations) is not currently used by Perl— except possibly to affect the behavior of library functions called by extensions outside the standard Perl distribution.

## SECURITY

Although the main discussion of Perl security issues can be found in the *perlsec* manpage, a discussion of Perl's locale handling would be incomplete if it did not draw your attention to locale-dependent security issues. Locales— particularly on systems that allow unprivileged users to build their own locales— are untrustworthy. A malicious (or just plain broken) locale can make a locale-aware application give unexpected results. Here are a few possibilities:

- Regular expression checks for safe file names or mail addresses using `\w` may be spoofed by an `LC_CTYPE` locale that claims that characters such as ">" and "|" are alphanumeric.
- String interpolation with case-mapping, as in, say, `$dest = "C:\U$name.$ext"`, may produce dangerous results if a bogus LC_CTYPE case-mapping table is in effect.
- If the decimal point character in the `LC_NUMERIC` locale is surreptitiously changed from a dot to a comma, `sprintf("%g", 0.123456e3)` produces a string result of "123,456". Many people would interpret this as one hundred and twenty-three thousand, four hundred and fifty-six.
- A sneaky `LC_COLLATE` locale could result in the names of students with "D" grades appearing ahead of those with "A"s.
- An application that takes the trouble to use information in `LC_MONETARY` may format debits as if they were credits and vice versa if that locale has been subverted. Or it might make payments in US dollars instead of Hong Kong dollars.
- The date and day names in dates formatted by *strftime()* could be manipulated to advantage by a malicious user able to subvert the `LC_DATE` locale. ("Look— it says I wasn't in the building on Sunday.")

Such dangers are not peculiar to the locale system: any aspect of an application's environment which may be modified maliciously presents similar challenges. Similarly, they are not specific to Perl: any programming language that allows you to write programs that take account of their environment exposes you to these issues.

Perl cannot protect you from all possibilities shown in the examples— there is no substitute for your own vigilance— but, when `use locale` is in effect, Perl uses the tainting mechanism (see the *perlsec* manpage) to mark string results that become locale-dependent, and which may be untrustworthy in consequence. Here is a summary of the tainting behavior of operators and functions that may be affected by the locale:

**Comparison operators** (`lt`, `le`, `ge`, `gt` and `cmp`):
Scalar true/false (or less/equal/greater) result is never tainted.

**Case-mapping interpolation** (with `\l`, `\L`, `\u` or `\U`)
Result string containing interpolated material is tainted if `use locale` is in effect.

**Matching operator** (`m//`):
Scalar true/false result never tainted.

Subpatterns, either delivered as a list-context result or as `$1` etc. are tainted if `use locale` is in effect, and the subpattern regular expression contains `\w` (to match an alphanumeric character), `\W` (non-alphanumeric character), `\s` (white-space character), or `\S` (non white-space character). The matched-pattern variable, $&, $` (pre-match), $' (post-match), and $+ (last match) are also tainted if `use locale` is in effect and the regular expression contains `\w`, `\W`, `\s`, or `\S`.

**Substitution operator** (`s///`):
Has the same behavior as the match operator. Also, the left operand of `=~` becomes tainted when `use locale` in effect if modified as a result of a substitution based on a regular expression match involving `\w`, `\W`, `\s`, or `\S`; or of case-mapping with `\l`, `\L`,`\u` or `\U`.

**In-memory formatting function** (sprintf()):
Result is tainted if ''use locale'' is in effect.

**Output formatting functions** (printf() and write()):
Success/failure result is never tainted.

**Case-mapping functions** (lc(), lcfirst(), uc(), ucfirst()):
Results are tainted if `use locale` is in effect.

**POSIX locale-dependent functions** (localeconv(), strcoll(), strftime(), strxfrm()):
Results are never tainted.

**POSIX character class tests** (isalnum(), isalpha(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isup- per(), isxdigit()):
True/false results are never tainted.

Three examples illustrate locale-dependent tainting. The first program, which ignores its locale, won't run: a value taken directly from the command line may not be used to name an output file when taint checks are enabled.

```
#/usr/local/bin/perl -T
# Run with taint checking

# Command line sanity check omitted...
$tainted_output_file = shift;

open(F, ">$tainted_output_file")
    or warn "Open of $untainted_output_file failed: $!\n";
```

The program can be made to run by ''laundering'' the tainted value through a regular expression: the second example— which still ignores locale information— runs, creating the file named on its command line if it can.

```
#/usr/local/bin/perl -T

$tainted_output_file = shift;
$tainted_output_file =~ m%[\w/]+%;
$untainted_output_file = $&;

open(F, ">$untainted_output_file")
    or warn "Open of $untainted_output_file failed: $!\n";
```

Compare this with a similar but locale-aware program:

```
#/usr/local/bin/perl -T

$tainted_output_file = shift;
use locale;
$tainted_output_file =~ m%[\w/]+%;
$localized_output_file = $&;
```

```
open(F, ">$localized_output_file")
    or warn "Open of $localized_output_file failed: $!\n";
```

This third program fails to run because $& is tainted: it is the result of a match involving `\w` while `use locale` is in effect.

## ENVIRONMENT

PERL_BADLANG
: A string that can suppress Perl's warning about failed locale settings at startup. Failure can occur if the locale support in the operating system is lacking (broken) in some way—or if you mistyped the name of a locale when you set up your environment. If this environment variable is absent, or has a value that does not evaluate to integer zero—that is, "0" or ""—Perl will complain about locale setting failures.

  **NOTE**: PERL_BADLANG only gives you a way to hide the warning message. The message tells about some problem in your system's locale support, and you should investigate what the problem is.

The following environment variables are not specific to Perl: They are part of the standardized (ISO C, XPG4, POSIX 1.c) *setlocale()* method for controlling an application's opinion on data.

LC_ALL
: `LC_ALL` is the ''override-all'' locale environment variable. If set, it overrides all the rest of the locale environment variables.

LC_CTYPE
: In the absence of `LC_ALL`, `LC_CTYPE` chooses the character type locale. In the absence of both `LC_ALL` and `LC_CTYPE`, `LANG` chooses the character type locale.

LC_COLLATE
: In the absence of `LC_ALL`, `LC_COLLATE` chooses the collation (sorting) locale. In the absence of both `LC_ALL` and `LC_COLLATE`, `LANG` chooses the collation locale.

LC_MONETARY
: In the absence of `LC_ALL`, `LC_MONETARY` chooses the monetary formatting locale. In the absence of both `LC_ALL` and `LC_MONETARY`, `LANG` chooses the monetary formatting locale.

LC_NUMERIC
: In the absence of `LC_ALL`, `LC_NUMERIC` chooses the numeric format locale. In the absence of both `LC_ALL` and `LC_NUMERIC`, `LANG` chooses the numeric format.

LC_TIME
: In the absence of `LC_ALL`, `LC_TIME` chooses the date and time formatting locale. In the absence of both `LC_ALL` and `LC_TIME`, `LANG` chooses the date and time formatting locale.

LANG
: `LANG` is the ''catch-all'' locale environment variable. If it is set, it is used as the last resort after the overall `LC_ALL` and the category-specific `LC_....`

## NOTES

### Backward compatibility

Versions of Perl prior to 5.004 **mostly** ignored locale information, generally behaving as if something similar to the `"C"` locale were always in force, even if the program environment suggested otherwise (see the section on *The setlocale function*). By default, Perl still behaves this way for backward compatibility. If you want a Perl application to pay attention to locale information, you **must** use the `use locale` pragma (see the section on *The use locale Pragma*) to instruct it to do so.

Versions of Perl from 5.002 to 5.003 did use the `LC_CTYPE` information if available; that is, `\w` did understand what were the letters according to the locale environment variables. The problem was that the user had no control over the feature: if the C library supported locales, Perl used them.

### I18N:Collate obsolete

In versions of Perl prior to 5.004, per-locale collation was possible using the `I18N::Collate` library module. This module is now mildly obsolete and should be avoided in new applications. The `LC_COLLATE` functionality is now integrated into the Perl core language: One can use locale-specific scalar data completely normally with `use locale`, so there is no longer any need to juggle with the scalar references of `I18N::Collate`.

### Sort speed and memory use impacts

Comparing and sorting by locale is usually slower than the default sorting; slow-downs of two to four times have been observed. It will also consume more memory: once a Perl scalar variable has participated in any string comparison or sorting operation obeying the locale collation rules, it will take 3-15 times more memory than before. (The exact multiplier depends on the string's contents, the operating system and the locale.) These downsides are dictated more by the operating system's implementation of the locale system than by Perl.

### *write()* and LC_NUMERIC

Formats are the only part of Perl that unconditionally use information from a program's locale; if a program's environment specifies an LC_NUMERIC locale, it is always used to specify the decimal point character in formatted output. Formatted output cannot be controlled by `use locale` because the pragma is tied to the block structure of the program, and, for historical reasons, formats exist outside that block structure.

### Freely available locale definitions

There is a large collection of locale definitions at `ftp://dkuug.dk/i18n/WG15-collection`. You should be aware that it is unsupported, and is not claimed to be fit for any purpose. If your system allows installation of arbitrary locales, you may find the definitions useful as they are, or as a basis for the development of your own locales.

### I18n and l10n

"Internationalization" is often abbreviated as **i18n** because its first and last letters are separated by eighteen others. (You may guess why the internalin ... internaliti ... i18n tends to get abbreviated.) In the same way, "localization" is often abbreviated to **l10n**.

### An imperfect standard

Internationalization, as defined in the C and POSIX standards, can be criticized as incomplete, ungainly, and having too large a granularity. (Locales apply to a whole process, when it would arguably be more useful to have them apply to a single thread, window group, or whatever.) They also have a tendency, like standards groups, to divide the world into nations, when we all know that the world can equally well be divided into bankers, bikers, gamers, and so on. But, for now, it's the only standard we've got. This may be construed as a bug.

## BUGS

### Broken systems

In certain systems, the operating system's locale support is broken and cannot be fixed or used by Perl. Such deficiencies can and will result in mysterious hangs and/or Perl core dumps when the `use locale` is in effect. When confronted with such a system, please report in excruciating detail to <*perlbug@perl.com*>, and complain to your vendor: bug fixes may exist for these problems in your operating system. Sometimes such bug fixes are called an operating system upgrade.

## SEE ALSO

the `isalnum` entry in the *POSIX (3)* manpage

the `isalpha` entry in the *POSIX (3)* manpage

the `isdigit` entry in the *POSIX (3)* manpage the `isgraph` entry in the *POSIX (3)* manpage

the `islower` entry in the *POSIX (3)* manpage

the `isprint` entry in the *POSIX (3)* manpage,

the `ispunct` entry in the *POSIX (3)* manpage

the `isspace` entry in the *POSIX (3)* manpage

the `isupper` entry in the *POSIX (3)* manpage,

the `isxdigit` entry in the *POSIX (3)* manpage

the `localeconv` entry in the *POSIX (3)* manpage

the `setlocale` entry in the *POSIX (3)* manpage,

the `strcoll` entry in the *POSIX (3)* manpage

the `strftime` entry in the *POSIX (3)* manpage

the `strtod` entry in the *POSIX (3)* manpage,

the `strxfrm` entry in the *POSIX (3)* manpage

**HISTORY**

Jarkko Hietaniemi's original *perli18n.pod* heavily hacked by Dominic Dunlop, assisted by the perl5-porters. Prose worked over a bit by Tom Christiansen.

Last update: Thu Jun 11 08:44:13 MDT 1998

**NAME**

perlref − Perl references and nested data structures

**DESCRIPTION**

Before release 5 of Perl it was difficult to represent complex data structures, because all references had to be symbolic— and even then it was difficult to refer to a variable instead of a symbol table entry. Perl now not only makes it easier to use symbolic references to variables, but also lets you have ''hard'' references to any piece of data or code. Any scalar may hold a hard reference. Because arrays and hashes contain scalars, you can now easily build arrays of arrays, arrays of hashes, hashes of arrays, arrays of hashes of functions, and so on.

Hard references are smart— they keep track of reference counts for you, automatically freeing the thing referred to when its reference count goes to zero. (Note: the reference counts for values in self-referential or cyclic data structures may not go to zero without a little help; see the section on *Two-Phased Garbage Collection* in the *perlobj* manpage for a detailed explanation.) If that thing happens to be an object, the object is destructed. See the *perlobj* manpage for more about objects. (In a sense, everything in Perl is an object, but we usually reserve the word for references to objects that have been officially ''blessed'' into a class package.)

Symbolic references are names of variables or other objects, just as a symbolic link in a Unix filesystem contains merely the name of a file. The `*glob` notation is a kind of symbolic reference. (Symbolic references are sometimes called ''soft references'', but please don't call them that; references are confusing enough without useless synonyms.)

In contrast, hard references are more like hard links in a Unix file system: They are used to access an underlying object without concern for what its (other) name is. When the word ''reference'' is used without an adjective, as in the following paragraph, it is usually talking about a hard reference.

References are easy to use in Perl. There is just one overriding principle: Perl does no implicit referencing or dereferencing. When a scalar is holding a reference, it always behaves as a simple scalar. It doesn't magically start being an array or hash or subroutine; you have to tell it explicitly to do so, by dereferencing it.

### Making References

References can be created in several ways.

1. By using the backslash operator on a variable, subroutine, or value. (This works much like the & (address-of) operator in C.) Note that this typically creates *ANOTHER* reference to a variable, because there's already a reference to the variable in the symbol table. But the symbol table reference might go away, and you'll still have the reference that the backslash returned. Here are some examples:

```
$scalarref = \$foo;
$arrayref  = \@ARGV;
$hashref   = \%ENV;
$coderef   = \&handler;
$globref   = \*foo;
```

   It isn't possible to create a true reference to an IO handle (filehandle or dirhandle) using the backslash operator. The most you can get is a reference to a typeglob, which is actually a complete symbol table entry. But see the explanation of the `*foo{THING}` syntax below. However, you can still use type globs and globrefs as though they were IO handles.

2. A reference to an anonymous array can be created using square brackets:

```
$arrayref = [1, 2, ['a', 'b', 'c']];
```

   Here we've created a reference to an anonymous array of three elements whose final element is itself a reference to another anonymous array of three elements. (The multidimensional syntax described later can be used to access this. For example, after the above, `$arrayref->[2][1]` would have the value "b".)

Note that taking a reference to an enumerated list is not the same as using square brackets—instead it's the same as creating a list of references!

```
@list = (\$a, \@b, \%c);
@list = \($a, @b, %c);      # same thing!
```

As a special case, `\(@foo)` returns a list of references to the contents of `@foo`, not a reference to `@foo` itself. Likewise for `%foo`.

3. A reference to an anonymous hash can be created using curly brackets:

```
$hashref = {
    'Adam'  => 'Eve',
    'Clyde' => 'Bonnie',
};
```

Anonymous hash and array composers like these can be intermixed freely to produce as complicated a structure as you want. The multidimensional syntax described below works for these too. The values above are literals, but variables and expressions would work just as well, because assignment operators in Perl (even within *local()* or *my()*) are executable statements, not compile-time declarations.

Because curly brackets (braces) are used for several other things including BLOCKs, you may occasionally have to disambiguate braces at the beginning of a statement by putting a + or a `return` in front so that Perl realizes the opening brace isn't starting a BLOCK. The economy and mnemonic value of using curlies is deemed worth this occasional extra hassle.

For example, if you wanted a function to make a new hash and return a reference to it, you have these options:

```
sub hashem {        { @_ } }   # silently wrong
sub hashem {       +{ @_ } }   # ok
sub hashem { return { @_ } }   # ok
```

On the other hand, if you want the other meaning, you can do this:

```
sub showem {        { @_ } }   # ambiguous (currently ok, but may change)
sub showem {       {; @_ } }   # ok
sub showem { { return @_ } }   # ok
```

Note how the leading `+{` and `{;` always serve to disambiguate the expression to mean either the HASH reference, or the BLOCK.

4. A reference to an anonymous subroutine can be created by using `sub` without a subname:

```
$coderef = sub { print "Boink!\n" };
```

Note the presence of the semicolon. Except for the fact that the code inside isn't executed immediately, a `sub {}` is not so much a declaration as it is an operator, like `do{}` or `eval{}`. (However, no matter how many times you execute that particular line (unless you're in an `eval("...")`), `$coderef` will still have a reference to the *SAME* anonymous subroutine.)

Anonymous subroutines act as closures with respect to *my()* variables, that is, variables visible lexically within the current scope. Closure is a notion out of the Lisp world that says if you define an anonymous function in a particular lexical context, it pretends to run in that context even when it's called outside of the context.

In human terms, it's a funny way of passing arguments to a subroutine when you define it as well as when you call it. It's useful for setting up little bits of code to run later, such as callbacks. You can even do object-oriented stuff with it, though Perl already provides a different mechanism to do that— see the *perlobj* manpage.

You can also think of closure as a way to write a subroutine template without using eval. (In fact, in version 5.000, eval was the *only* way to get closures. You may wish to use “require 5.001” if you use closures.)

Here’s a small example of how closures works:

```
sub newprint {
    my $x = shift;
    return sub { my $y = shift; print "$x, $y!\n"; };
}
$h = newprint("Howdy");
$g = newprint("Greetings");

# Time passes...

&$h("world");
&$g("earthlings");
```

This prints

```
Howdy, world!
Greetings, earthlings!
```

Note particularly that `$x` continues to refer to the value passed into *newprint() despite* the fact that the “my `$x`” has seemingly gone out of scope by the time the anonymous subroutine runs. That’s what closure is all about.

This applies only to lexical variables, by the way. Dynamic variables continue to work as they have always worked. Closure is not something that most Perl programmers need trouble themselves about to begin with.

5. References are often returned by special subroutines called constructors. Perl objects are just references to a special kind of object that happens to know which package it’s associated with. Constructors are just special subroutines that know how to create that association. They do so by starting with an ordinary reference, and it remains an ordinary reference even while it’s also being an object. Constructors are often named *new()* and called indirectly:

```
$objref = new Doggie (Tail => 'short', Ears => 'long');
```

But don’t have to be:

```
$objref   = Doggie->new(Tail => 'short', Ears => 'long');

use Term::Cap;
$terminal = Term::Cap->Tgetent( { OSPEED => 9600 });

use Tk;
$main    = MainWindow->new();
$menubar = $main->Frame(-relief              => "raised",
                        -borderwidth         => 2)
```

6. References of the appropriate type can spring into existence if you dereference them in a context that assumes they exist. Because we haven’t talked about dereferencing yet, we can’t show you any examples yet.

7. A reference can be created by using a special syntax, lovingly known as the *foo{THING} syntax. *foo{THING} returns a reference to the THING slot in *foo (which is the symbol table entry which holds everything known as foo).

```
$scalarref = *foo{SCALAR};
$arrayref  = *ARGV{ARRAY};
$hashref   = *ENV{HASH};
$coderef   = *handler{CODE};
$ioref     = *STDIN{IO};
$globref   = *foo{GLOB};
```

All of these are self-explanatory except for *foo{IO}. It returns the IO handle, used for file handles (the `open` entry in the *perlfunc* manpage), sockets (the `socket` entry in the *perlfunc* manpage and the `socketpair` entry in the *perlfunc* manpage), and directory handles (the `opendir` entry in the *perlfunc* manpage). For compatibility with previous versions of Perl, *foo{FILEHANDLE} is a synonym for *foo{IO}.

*foo{THING} returns undef if that particular THING hasn't been used yet, except in the case of scalars. *foo{SCALAR} returns a reference to an anonymous scalar if `$foo` hasn't been used yet. This might change in a future release.

*foo{IO} is an alternative to the *HANDLE mechanism given in the section on *Typeglobs and Filehandles* in the *perldata* manpage for passing filehandles into or out of subroutines, or storing into larger data structures. Its disadvantage is that it won't create a new filehandle for you. Its advantage is that you have no risk of clobbering more than you want to with a typeglob assignment, although if you assign to a scalar instead of a typeglob, you're ok.

```
splutter(*STDOUT);
splutter(*STDOUT{IO});

sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}

$rec = get_rec(*STDIN);
$rec = get_rec(*STDIN{IO});

sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

**Using References**

That's it for creating references. By now you're probably dying to know how to use references to get back to your long-lost data. There are several basic methods.

1. Anywhere you'd put an identifier (or chain of identifiers) as part of a variable or subroutine name, you can replace the identifier with a simple scalar variable containing a reference of the correct type:

```
$bar = $$scalarref;
push(@$arrayref, $filename);
$$arrayref[0] = "January";
$$hashref{"KEY"} = "VALUE";
&$coderef(1,2,3);
print $globref "output\n";
```

It's important to understand that we are specifically *NOT* dereferencing `$arrayref[0]` or `$hashref{"KEY"}` there. The dereference of the scalar variable happens *BEFORE* it does any key lookups. Anything more complicated than a simple scalar variable must use methods 2 or 3 below. However, a "simple scalar" includes an identifier that itself uses method 1 recursively. Therefore, the following prints "howdy".

```
$refrefref = \\\"howdy";
print $$$$refrefref;
```

2. Anywhere you'd put an identifier (or chain of identifiers) as part of a variable or subroutine name, you can replace the identifier with a BLOCK returning a reference of the correct type. In other words, the previous examples could be written like this:

```
$bar = ${$scalarref};
push(@{$arrayref}, $filename);
${$arrayref}[0] = "January";
${$hashref}{"KEY"} = "VALUE";
&{$coderef}(1,2,3);
$globref->print("output\n");  # iff IO::Handle is loaded
```

Admittedly, it's a little silly to use the curlies in this case, but the BLOCK can contain any arbitrary expression, in particular, subscripted expressions:

```
&{ $dispatch{$index} }(1,2,3);      # call correct routine
```

Because of being able to omit the curlies for the simple case of `$$x`, people often make the mistake of viewing the dereferencing symbols as proper operators, and wonder about their precedence. If they were, though, you could use parentheses instead of braces. That's not the case. Consider the difference below; case 0 is a short-hand version of case 1, *NOT* case 2:

```
$$hashref{"KEY"}   = "VALUE";       # CASE 0
${$hashref}{"KEY"} = "VALUE";       # CASE 1
${$hashref{"KEY"}} = "VALUE";       # CASE 2
${$hashref->{"KEY"}} = "VALUE";     # CASE 3
```

Case 2 is also deceptive in that you're accessing a variable called `%hashref`, not dereferencing through `$hashref` to the hash it's presumably referencing. That would be case 3.

3. Subroutine calls and lookups of individual array elements arise often enough that it gets cumbersome to use method 2. As a form of syntactic sugar, the examples for method 2 may be written:

```
$arrayref->[0] = "January";   # Array element
$hashref->{"KEY"} = "VALUE";  # Hash element
$coderef->(1,2,3);            # Subroutine call
```

The left side of the arrow can be any expression returning a reference, including a previous dereference. Note that `$array[$x]` is *NOT* the same thing as `$array->[$x]` here:

```
$array[$x]->{"foo"}->[0] = "January";
```

This is one of the cases we mentioned earlier in which references could spring into existence when in an lvalue context. Before this statement, `$array[$x]` may have been undefined. If so, it's automatically defined with a hash reference so that we can look up `{"foo"}` in it. Likewise `$array[$x]->{"foo"}` will automatically get defined with an array reference so that we can look up `[0]` in it. This process is called *autovivification*.

One more thing here. The arrow is optional *BETWEEN* brackets subscripts, so you can shrink the above down to

```
$array[$x]{"foo"}[0] = "January";
```

Which, in the degenerate case of using only ordinary arrays, gives you multidimensional arrays just like C's:

```
$score[$x][$y][$z] += 42;
```

Well, okay, not entirely like C's arrays, actually. C doesn't know how to grow its arrays on demand. Perl does.

4. If a reference happens to be a reference to an object, then there are probably methods to access the things referred to, and you should probably stick to those methods unless you're in the class package that defines the object's methods. In other words, be nice, and don't violate the object's encapsulation without a very good reason. Perl does not enforce encapsulation. We are not totalitarians here. We do expect some basic civility though.

The *ref()* operator may be used to determine what type of thing the reference is pointing to. See the *perlfunc* manpage.

The *bless()* operator may be used to associate the object a reference points to with a package functioning as an object class. See the *perlobj* manpage.

A typeglob may be dereferenced the same way a reference can, because the dereference syntax always indicates the kind of reference desired. So `${*foo}` and `${\$foo}` both indicate the same scalar variable.

Here's a trick for interpolating a subroutine call into a string:

```
print "My sub returned @{[mysub(1,2,3)]} that time.\n";
```

The way it works is that when the `@{...}` is seen in the double-quoted string, it's evaluated as a block. The block creates a reference to an anonymous array containing the results of the call to `mysub(1,2,3)`. So the whole block returns a reference to an array, which is then dereferenced by `@{...}` and stuck into the double-quoted string. This chicanery is also useful for arbitrary expressions:

```
print "That yields @{[$n + 5]} widgets\n";
```

**Symbolic references**

We said that references spring into existence as necessary if they are undefined, but we didn't say what happens if a value used as a reference is already defined, but *ISN'T* a hard reference. If you use it as a reference in this case, it'll be treated as a symbolic reference. That is, the value of the scalar is taken to be the *NAME* of a variable, rather than a direct link to a (possibly) anonymous value.

People frequently expect it to work like this. So it does.

```
$name = "foo";
$$name = 1;                 # Sets $foo
${$name} = 2;               # Sets $foo
${$name x 2} = 3;           # Sets $foofoo
$name->[0] = 4;             # Sets $foo[0]
@$name = ();                # Clears @foo
&$name();                   # Calls &foo() (as in Perl 4)
$pack = "THAT";
${"${pack}::$name"} = 5;    # Sets $THAT::foo without eval
```

This is very powerful, and slightly dangerous, in that it's possible to intend (with the utmost sincerity) to use a hard reference, and accidentally use a symbolic reference instead. To protect against that, you can say

```
use strict 'refs';
```

and then only hard references will be allowed for the rest of the enclosing block. An inner block may countermand that with

```
no strict 'refs';
```

Only package variables (globals, even if localized) are visible to symbolic references. Lexical variables (declared with *my()*) aren't in a symbol table, and thus are invisible to this mechanism. For example:

```
local $value = 10;
$ref = \$value;
{
    my $value = 20;
    print $$ref;
}
```

This will still print 10, not 20. Remember that *local()* affects package variables, which are all "global" to the package.

**Not-so-symbolic references**

A new feature contributing to readability in perl version 5.001 is that the brackets around a symbolic reference behave more like quotes, just as they always have within a string. That is,

```
$push = "pop on ";
print "${push}over";
```

has always meant to print "pop on over", despite the fact that push is a reserved word. This has been generalized to work the same outside of quotes, so that

```
print ${push} . "over";
```

and even

```
print ${ push } . "over";
```

will have the same effect. (This would have been a syntax error in Perl 5.000, though Perl 4 allowed it in the spaceless form.) Note that this construct is *not* considered to be a symbolic reference when you're using strict refs:

```
use strict 'refs';
${ bareword };      # Okay, means $bareword.
${ "bareword" };    # Error, symbolic reference.
```

Similarly, because of all the subscripting that is done using single words, we've applied the same rule to any bareword that is used for subscripting a hash. So now, instead of writing

```
$array{ "aaa" }{ "bbb" }{ "ccc" }
```

you can write just

```
$array{ aaa }{ bbb }{ ccc }
```

and not worry about whether the subscripts are reserved words. In the rare event that you do wish to do something like

```
$array{ shift }
```

you can force interpretation as a reserved word by adding anything that makes it more than a bareword:

```
$array{ shift() }
$array{ +shift }
$array{ shift @_ }
```

The **−w** switch will warn you if it interprets a reserved word as a string. But it will no longer warn you about using lowercase words, because the string is effectively quoted.

### Pseudo-hashes: Using an array as a hash

WARNING: This section describes an experimental feature. Details may change without notice in future versions.

Beginning with release 5.005 of Perl you can use an array reference in some contexts that would normally require a hash reference. This allows you to access array elements using symbolic names, as if they were fields in a structure.

For this to work, the array must contain extra information. The first element of the array has to be a hash reference that maps field names to array indices. Here is an example:

```
$struct = [{foo => 1, bar => 2}, "FOO", "BAR"];

$struct->{foo};  # same as $struct->[1], i.e. "FOO"
$struct->{bar};  # same as $struct->[2], i.e. "BAR"

keys %$struct;   # will return ("foo", "bar") in some order
values %$struct; # will return ("FOO", "BAR") in same some order

while (my($k,$v) = each %$struct) {
    print "$k => $v\n";
}
```

Perl will raise an exception if you try to delete keys from a pseudo-hash or try to access nonexistent fields. For better performance, Perl can also do the translation from field names to array indices at compile time for typed object references. See the *fields* manpage.

### Function Templates

As explained above, a closure is an anonymous function with access to the lexical variables visible when that function was compiled. It retains access to those variables even though it doesn't get run until later, such as in a signal handler or a Tk callback.

Using a closure as a function template allows us to generate many functions that act similarly. Suppopose you wanted functions named after the colors that generated HTML font changes for the various colors:

```
print "Be ", red("careful"), "with that ", green("light");
```

The *red()* and *green()* functions would be very similar. To create these, we'll assign a closure to a typeglob of the name of the function we're trying to build.

```
@colors = qw(red blue green yellow orange purple violet);
for my $name (@colors) {
    no strict 'refs';       # allow symbol table manipulation
    *$name = *{uc $name} = sub { "<FONT COLOR='$name'>@_</FONT>" };
}
```

Now all those different functions appear to exist independently. You can call *red()*, *RED()*, *blue()*, *BLUE()*, *green()*, etc. This technique saves on both compile time and memory use, and is less error-prone as well, since syntax checks happen at compile time. It's critical that any variables in the anonymous subroutine be lexicals in order to create a proper closure. That's the reasons for the `my` on the loop iteration variable.

This is one of the only places where giving a prototype to a closure makes much sense. If you wanted to impose scalar context on the arguments of these functions (probably not a wise idea for this particular example), you could have written it this way instead:

```
*$name = sub ($) { "<FONT COLOR='$name'>$_[0]</FONT>" };
```

However, since prototype checking happens at compile time, the assignment above happens too late to be of much use. You could address this by putting the whole loop of assignments within a BEGIN block, forcing it to occur during compilation.

Access to lexicals that change over type— like those in the `for` loop above— only works with closures, not general subroutines. In the general case, then, named subroutines do not nest properly, although anonymous ones do. If you are accustomed to using nested subroutines in other programming languages with their own private variables, you'll have to work at it a bit in Perl. The intuitive coding of this kind of thing incurs mysterious warnings about "will not stay shared". For example, this won't work:

```
sub outer {
    my $x = $_[0] + 35;
    sub inner { return $x * 19 }   # WRONG
    return $x + inner();
}
```

A work-around is the following:

```
sub outer {
    my $x = $_[0] + 35;
    local *inner = sub { return $x * 19 };
    return $x + inner();
}
```

Now *inner()* can only be called from within *outer()*, because of the temporary assignments of the closure (anonymous subroutine). But when it does, it has normal access to the lexical variable `$x` from the scope of *outer()*.

This has the interesting effect of creating a function local to another function, something not normally supported in Perl.

**WARNING**

You may not (usefully) use a reference as the key to a hash. It will be converted into a string:

```
$x{ \$a } = $a;
```

If you try to dereference the key, it won't do a hard dereference, and you won't accomplish what you're attempting. You might want to do something more like

```
$r = \@a;
$x{ $r } = $r;
```

And then at least you can use the *values()*, which will be real refs, instead of the *keys()*, which won't.

The standard Tie::RefHash module provides a convenient workaround to this.

**SEE ALSO**

Besides the obvious documents, source code can be instructive. Some rather pathological examples of the use of references can be found in the *t/op/ref.t* regression test in the Perl source directory.

See also the *perldsc* manpage and the *perllol* manpage for how to use references to create complex data structures, and the *perltoot* manpage, the *perlobj* manpage, and the *perlbot* manpage for how to use them to create objects.

**NAME**

perldsc − Perl Data Structures Cookbook

**DESCRIPTION**

The single feature most sorely lacking in the Perl programming language prior to its 5.0 release was complex data structures. Even without direct language support, some valiant programmers did manage to emulate them, but it was hard work and not for the faint of heart. You could occasionally get away with the `$m{$LoL,$b}` notation borrowed from *awk* in which the keys are actually more like a single concatenated string `"$LoL$b"`, but traversal and sorting were difficult. More desperate programmers even hacked Perl's internal symbol table directly, a strategy that proved hard to develop and maintain— to put it mildly.

The 5.0 release of Perl let us have complex data structures. You may now write something like this and all of a sudden, you'd have a array with three dimensions!

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        for $z (1 .. 10) {
            $LoL[$x][$y][$z] =
                $x ** $y + $z;
        }
    }
}
```

Alas, however simple this may appear, underneath it's a much more elaborate construct than meets the eye!

How do you print it out? Why can't you say just `print @LoL`? How do you sort it? How can you pass it to a function or get one of these back from a function? Is is an object? Can you save it to disk to read back later? How do you access whole rows or columns of that matrix? Do all the values have to be numeric?

As you see, it's quite easy to become confused. While some small portion of the blame for this can be attributed to the reference-based implementation, it's really more due to a lack of existing documentation with examples designed for the beginner.

This document is meant to be a detailed but understandable treatment of the many different sorts of data structures you might want to develop. It should also serve as a cookbook of examples. That way, when you need to create one of these complex data structures, you can just pinch, pilfer, or purloin a drop-in example from here.

Let's look at each of these possible constructs in detail. There are separate sections on each of the following:

• arrays of arrays

• hashes of arrays

• arrays of hashes

• hashes of hashes

• more elaborate constructs

But for now, let's look at general issues common to all these types of data structures.

**REFERENCES**

The most important thing to understand about all data structures in Perl -- including multidimensional arrays— is that even though they might appear otherwise, Perl `@ARRAYs` and `%HASHes` are all internally one-dimensional. They can hold only scalar values (meaning a string, number, or a reference). They cannot directly contain other arrays or hashes, but instead contain *references* to other arrays or hashes.

You can't use a reference to a array or hash in quite the same way that you would a real array or hash. For C or C++ programmers unused to distinguishing between arrays and pointers to the same, this can be confusing. If so, just think of it as the difference between a structure and a pointer to a structure.

You can (and should) read more about references in the *perlref*(1) man page. Briefly, references are rather like pointers that know what they point to. (Objects are also a kind of reference, but we won't be needing them right away— if ever.) This means that when you have something which looks to you like an access to a two-or-more-dimensional array and/or hash, what's really going on is that the base type is merely a one-dimensional entity that contains references to the next level. It's just that you can *use* it as though it were a two-dimensional one. This is actually the way almost all C multidimensional arrays work as well.

```
    $list[7][12]                        # array of arrays
    $list[7]{string}                    # array of hashes
    $hash{string}[7]                    # hash of arrays
    $hash{string}{'another string'}     # hash of hashes
```

Now, because the top level contains only references, if you try to print out your array in with a simple *print()* function, you'll get something that doesn't look very nice, like this:

```
    @LoL = ( [2, 3], [4, 5, 7], [0] );
    print $LoL[1][2];
  7
    print @LoL;
  ARRAY(0x83c38)ARRAY(0x8b194)ARRAY(0x8b1d0)
```

That's because Perl doesn't (ever) implicitly dereference your variables. If you want to get at the thing a reference is referring to, then you have to do this yourself using either prefix typing indicators, like `${$blah}`, `@{$blah}`, `@{$blah[$i]}`, or else postfix pointer arrows, like `$a->[3]`, `$h->{fred}`, or even `$ob->method()->[3]`.

## COMMON MISTAKES

The two most common mistakes made in constructing something like an array of arrays is either accidentally counting the number of elements or else taking a reference to the same memory location repeatedly. Here's the case where you just get the count instead of a nested array:

```
    for $i (1..10) {
        @list = somefunc($i);
        $LoL[$i] = @list;       # WRONG!
    }
```

That's just the simple case of assigning a list to a scalar and getting its element count. If that's what you really and truly want, then you might do well to consider being a tad more explicit about it, like this:

```
    for $i (1..10) {
        @list = somefunc($i);
        $counts[$i] = scalar @list;
    }
```

Here's the case of taking a reference to the same memory location again and again:

```
    for $i (1..10) {
        @list = somefunc($i);
        $LoL[$i] = \@list;      # WRONG!
    }
```

So, what's the big problem with that? It looks right, doesn't it? After all, I just told you that you need an array of references, so by golly, you've made me one!

Unfortunately, while this is true, it's still broken. All the references in `@LoL` refer to the *very same place*, and they will therefore all hold whatever was last in `@list`! It's similar to the problem demonstrated in the following C program:

```
#include <pwd.h>
main() {
    struct passwd *getpwnam(), *rp, *dp;
    rp = getpwnam("root");
    dp = getpwnam("daemon");

    printf("daemon name is %s\nroot name is %s\n",
            dp->pw_name, rp->pw_name);
}
```

Which will print

```
daemon name is daemon
root name is daemon
```

The problem is that both `rp` and `dp` are pointers to the same location in memory! In C, you'd have to remember to *malloc()* yourself some new memory. In Perl, you'll want to use the array constructor `[]` or the hash constructor `{}` instead. Here's the right way to do the preceding broken code fragments:

```
for $i (1..10) {
    @list = somefunc($i);
    $LoL[$i] = [ @list ];
}
```

The square brackets make a reference to a new array with a *copy* of what's in `@list` at the time of the assignment. This is what you want.

Note that this will produce something similar, but it's much harder to read:

```
for $i (1..10) {
    @list = 0 .. $i;
    @{$LoL[$i]} = @list;
}
```

Is it the same? Well, maybe so— and maybe not. The subtle difference is that when you assign something in square brackets, you know for sure it's always a brand new reference with a new *copy* of the data. Something else could be going on in this new case with the `@{$LoL[$i]}` dereference on the left-hand-side of the assignment. It all depends on whether `$LoL[$i]` had been undefined to start with, or whether it already contained a reference. If you had already populated `@LoL` with references, as in

```
$LoL[3] = \@another_list;
```

Then the assignment with the indirection on the left-hand-side would use the existing reference that was already there:

```
@{$LoL[3]} = @list;
```

Of course, this *would* have the "interesting" effect of clobbering `@another_list`. (Have you ever noticed how when a programmer says something is "interesting", that rather than meaning "intriguing", they're disturbingly more apt to mean that it's "annoying", "difficult", or both? :–)

So just remember always to use the array or hash constructors with `[]` or `{}`, and you'll be fine, although it's not always optimally efficient.

Surprisingly, the following dangerous-looking construct will actually work out fine:

```
for $i (1..10) {
    my @list = somefunc($i);
    $LoL[$i] = \@list;
}
```

That's because *my()* is more of a run-time statement than it is a compile-time declaration *per se*. This means that the *my()* variable is remade afresh each time through the loop. So even though it *looks* as though you stored the same variable reference each time, you actually did not! This is a subtle distinction that can produce more efficient code at the risk of misleading all but the most experienced of programmers. So I usually advise against teaching it to beginners. In fact, except for passing arguments to functions, I seldom like to see the gimme-a-reference operator (backslash) used much at all in code. Instead, I advise beginners that they (and most of the rest of us) should try to use the much more easily understood constructors `[]` and `{}` instead of relying upon lexical (or dynamic) scoping and hidden reference-counting to do the right thing behind the scenes.

In summary:

```
$LoL[$i] = [ @list ];       # usually best
$LoL[$i] = \@list;          # perilous; just how my() was that list?
@{ $LoL[$i] } = @list;      # way too tricky for most programmers
```

**CAVEAT ON PRECEDENCE**

Speaking of things like `@{$LoL[$i]}`, the following are actually the same thing:

```
$listref->[2][2]    # clear
$$listref[2][2]     # confusing
```

That's because Perl's precedence rules on its five prefix dereferencers (which look like someone swearing: `$ @ * % &`) make them bind more tightly than the postfix subscripting brackets or braces! This will no doubt come as a great shock to the C or C++ programmer, who is quite accustomed to using `*a[i]` to mean what's pointed to by the *i'th* element of `a`. That is, they first take the subscript, and only then dereference the thing at that subscript. That's fine in C, but this isn't C.

The seemingly equivalent construct in Perl, `$$listref[$i]` first does the deref of `$listref`, making it take `$listref` as a reference to an array, and then dereference that, and finally tell you the *i'th* value of the array pointed to by `$LoL`. If you wanted the C notion, you'd have to write `${$LoL[$i]}` to force the `$LoL[$i]` to get evaluated first before the leading `$` dereferencer.

**WHY YOU SHOULD ALWAYS** `use strict`

If this is starting to sound scarier than it's worth, relax. Perl has some features to help you avoid its most common pitfalls. The best way to avoid getting confused is to start every program like this:

```
#!/usr/bin/perl -w
use strict;
```

This way, you'll be forced to declare all your variables with *my()* and also disallow accidental "symbolic dereferencing". Therefore if you'd done this:

```
my $listref = [
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $listref[2][2];
```

The compiler would immediately flag that as an error *at compile time*, because you were accidentally accessing `@listref`, an undeclared variable, and it would thereby remind you to write instead:

```
    print $listref->[2][2]
```

## DEBUGGING

Before version 5.002, the standard Perl debugger didn't do a very nice job of printing out complex data structures. With 5.002 or above, the debugger includes several new features, including command line editing as well as the `x` command to dump out complex data structures. For example, given the assignment to `$LoL` above, here's the debugger output:

```
    DB<1> x $LoL
    $LoL = ARRAY(0x13b5a0)
       0  ARRAY(0x1f0a24)
          0  'fred'
          1  'barney'
          2  'pebbles'
          3  'bambam'
          4  'dino'
       1  ARRAY(0x13b558)
          0  'homer'
          1  'bart'
          2  'marge'
          3  'maggie'
       2  ARRAY(0x13b540)
          0  'george'
          1  'jane'
          2  'elroy'
          3  'judy'
```

## CODE EXAMPLES

Presented with little comment (these will get their own manpages someday) here are short code examples illustrating access of various types of data structures.

## LISTS OF LISTS

### Declaration of a LIST OF LISTS

```
 @LoL = (
        [ "fred", "barney" ],
        [ "george", "jane", "elroy" ],
        [ "homer", "marge", "bart" ],
      );
```

### Generation of a LIST OF LISTS

```
 # reading from file
 while ( <> ) {
     push @LoL, [ split ];
 }

 # calling a function
 for $i ( 1 .. 10 ) {
     $LoL[$i] = [ somefunc($i) ];
 }
```

```
# using temp vars
for $i ( 1 .. 10 ) {
    @tmp = somefunc($i);
    $LoL[$i] = [ @tmp ];
}

# add to an existing row
push @{ $LoL[0] }, "wilma", "betty";
```

### Access and Printing of a LIST OF LISTS

```
# one element
$LoL[0][0] = "Fred";

# another element
$LoL[1][1] =~ s/(\w)/\u$1/;

# print the whole thing with refs
for $aref ( @LoL ) {
    print "\t [ @$aref ],\n";
}

# print the whole thing with indices
for $i ( 0 .. $#LoL ) {
    print "\t [ @{$LoL[$i]} ],\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#LoL ) {
    for $j ( 0 .. $#{ $LoL[$i] } ) {
        print "elt $i $j is $LoL[$i][$j]\n";
    }
}
```

## HASHES OF LISTS

### Declaration of a HASH OF LISTS

```
%HoL = (
       flintstones        => [ "fred", "barney" ],
       jetsons            => [ "george", "jane", "elroy" ],
       simpsons           => [ "homer", "marge", "bart" ],
     );
```

### Generation of a HASH OF LISTS

```
# reading from file
# flintstones: fred barney wilma dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $HoL{$1} = [ split ];
}
```

```
# reading from file; more temps
# flintstones: fred barney wilma dino
while ( $line = <> ) {
    ($who, $rest) = split /:\s*/, $line, 2;
    @fields = split ' ', $rest;
    $HoL{$who} = [ @fields ];
}

# calling a function that returns a list
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoL{$group} = [ get_family($group) ];
}

# likewise, but using temps
for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoL{$group} = [ @members ];
}

# append new members to an existing family
push @{ $HoL{"flintstones"} }, "wilma", "betty";
```

### Access and Printing of a HASH OF LISTS

```
# one element
$HoL{flintstones}[0] = "Fred";

# another element
$HoL{simpsons}[1] =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoL ) {
    print "$family: @{ $HoL{$family} }\n"
}

# print the whole thing with indices
foreach $family ( keys %HoL ) {
    print "family: ";
    foreach $i ( 0 .. $#{ $HoL{$family} } ) {
        print " $i = $HoL{$family}[$i]";
    }
    print "\n";
}

# print the whole thing sorted by number of members
foreach $family ( sort { @{$HoL{$b}} <=> @{$HoL{$a}} } keys %HoL ) {
    print "$family: @{ $HoL{$family} }\n"
}
```

```
# print the whole thing sorted by number of members and name
foreach $family ( sort {
                            @{$HoL{$b}} <=> @{$HoL{$a}}
                                        ||
                                    $a cmp $b
           } keys %HoL )
{
    print "$family: ", join(", ", sort @{ $HoL{$family} }), "\n";
}
```

## LISTS OF HASHES

### Declaration of a LIST OF HASHES

```
@LoH = (
       {
           Lead     => "fred",
           Friend   => "barney",
       },
       {
           Lead     => "george",
           Wife     => "jane",
           Son      => "elroy",
       },
       {
           Lead     => "homer",
           Wife     => "marge",
           Son      => "bart",
       }
 );
```

### Generation of a LIST OF HASHES

```
# reading from file
# format: LEAD=fred FRIEND=barney
while ( <> ) {
    $rec = {};
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
    push @LoH, $rec;
}

# reading from file
# format: LEAD=fred FRIEND=barney
# no temp
while ( <> ) {
    push @LoH, { split /[\s+=]/ };
}
```

```
# calling a function  that returns a key,value list, like
# "lead","fred","daughter","pebbles"
while ( %fields = getnextpairset() ) {
    push @LoH, { %fields };
}

# likewise, but using no temp vars
while (<>) {
    push @LoH, { parsepairs($_) };
}

# add key/value to an element
$LoH[0]{pet} = "dino";
$LoH[2]{pet} = "santa's little helper";
```

### Access and Printing of a LIST OF HASHES

```
# one element
$LoH[0]{lead} = "fred";

# another element
$LoH[1]{lead} =~ s/(\w)/\u$1/;

# print the whole thing with refs
for $href ( @LoH ) {
    print "{ ";
    for $role ( keys %$href ) {
        print "$role=$href->{$role} ";
    }
    print "}\n";
}

# print the whole thing with indices
for $i ( 0 .. $#LoH ) {
    print "$i is { ";
    for $role ( keys %{ $LoH[$i] } ) {
        print "$role=$LoH[$i]{$role} ";
    }
    print "}\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#LoH ) {
    for $role ( keys %{ $LoH[$i] } ) {
        print "elt $i $role is $LoH[$i]{$role}\n";
    }
}
```

## HASHES OF HASHES

### Declaration of a HASH OF HASHES

```
%HoH = (
       flintstones => {
               lead      => "fred",
               pal       => "barney",
       },
       jetsons     => {
               lead      => "george",
               wife      => "jane",
               "his boy" => "elroy",
       },
       simpsons    => {
               lead      => "homer",
               wife      => "marge",
               kid       => "bart",
       },
);
```

### Generation of a HASH OF HASHES

```
# reading from file
# flintstones: lead=fred pal=barney wife=wilma pet=dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $HoH{$who}{$key} = $value;
    }

# reading from file; more temps
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    $rec = {};
    $HoH{$who} = $rec;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
}

# calling a function  that returns a key,value hash
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoH{$group} = { get_family($group) };
}

# likewise, but using temps
for $group ( "simpsons", "jetsons", "flintstones" ) {
    %members = get_family($group);
    $HoH{$group} = { %members };
}
```

```
# append new members to an existing family
%new_folks = (
    wife => "wilma",
    pet  => "dino",
);

for $what (keys %new_folks) {
    $HoH{flintstones}{$what} = $new_folks{$what};
}
```

### Access and Printing of a HASH OF HASHES

```
# one element
$HoH{flintstones}{wife} = "wilma";

# another element
$HoH{simpsons}{lead} =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoH ) {
    print "$family: { ";
    for $role ( keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# print the whole thing  somewhat sorted
foreach $family ( sort keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# print the whole thing sorted by number of members
foreach $family ( sort { keys %{$HoH{$b}} <=> keys %{$HoH{$a}} } keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# establish a sort order (rank) for each role
$i = 0;
for ( qw(lead wife son daughter pal pet) ) { $rank{$_} = ++$i }
```

```
# now print the whole thing sorted by number of members
foreach $family ( sort { keys %{ $HoH{$b} } <=> keys %{ $HoH{$a} } } keys %HoH ) {
    print "$family: { ";
    # and print these according to rank order
    for $role ( sort { $rank{$a} <=> $rank{$b} }  keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}
```

## MORE ELABORATE RECORDS

### Declaration of MORE ELABORATE RECORDS

Here's a sample showing how to create and use a record whose fields are of many different sorts:

```
$rec = {
    TEXT      => $string,
    SEQUENCE  => [ @old_values ],
    LOOKUP    => { %some_table },
    THATCODE  => \&some_function,
    THISCODE  => sub { $_[0] ** $_[1] },
    HANDLE    => \*STDOUT,
};

print $rec->{TEXT};

print $rec->{LIST}[0];
$last = pop @ { $rec->{SEQUENCE} };

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} };

$answer = $rec->{THATCODE}->($arg);
$answer = $rec->{THISCODE}->($arg1, $arg2);

# careful of extra block braces on fh ref
print { $rec->{HANDLE} } "a string\n";

use FileHandle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print(" a string\n");
```

### Declaration of a HASH OF COMPLEX RECORDS

```
%TV = (
   flintstones => {
       series   => "flintstones",
       nights   => [ qw(monday thursday friday) ],
       members  => [
           { name => "fred",    role => "lead", age  => 36, },
           { name => "wilma",   role => "wife", age  => 31, },
           { name => "pebbles", role => "kid",  age  =>  4, },
       ],
   },
```

```
    jetsons     => {
        series   => "jetsons",
        nights   => [ qw(wednesday saturday) ],
        members  => [
            { name => "george",  role => "lead", age  => 41, },
            { name => "jane",    role => "wife", age  => 39, },
            { name => "elroy",   role => "kid",  age  =>  9, },
        ],
     },

    simpsons    => {
        series   => "simpsons",
        nights   => [ qw(monday) ],
        members  => [
            { name => "homer", role => "lead", age  => 34, },
            { name => "marge", role => "wife", age => 37, },
            { name => "bart",  role => "kid",  age  =>  11, },
        ],
     },
  );
```

**Generation of a HASH OF COMPLEX RECORDS**

```
# reading from file
# this is most easily done by having the file itself be
# in the raw data format as shown above.  perl is happy
# to parse complex data structures if declared as data, so
# sometimes it's easiest to do that

# here's a piece by piece build up
$rec = {};
$rec->{series} = "flintstones";
$rec->{nights} = [ find_days() ];

@members = ();
# assume this file in field=value syntax
while (<>) {
    %fields = split /[\s=]+/;
    push @members, { %fields };
}
$rec->{members} = [ @members ];

# now remember the whole thing
$TV{ $rec->{series} } = $rec;
```

```
###########################################################
# now, you might want to make interesting extra fields that
# include pointers back into the same data structure so if
# change one piece, it changes everywhere, like for examples
# if you wanted a {kids} field that was an array reference
# to a list of the kids' records without having duplicate
# records and thus update problems.
###########################################################
foreach $family (keys %TV) {
    $rec = $TV{$family}; # temp pointer
    @kids = ();
    for $person ( @{ $rec->{members} } ) {
        if ($person->{role} =~ /kid|son|daughter/) {
            push @kids, $person;
        }
    }
    # REMEMBER: $rec and $TV{$family} point to same data!!
    $rec->{kids} = [ @kids ];
}

# you copied the list, but the list itself contains pointers
# to uncopied objects. this means that if you make bart get
# older via

$TV{simpsons}{kids}[0]{age}++;

# then this would also change in
print $TV{simpsons}{members}[2]{age};

# because $TV{simpsons}{kids}[0] and $TV{simpsons}{members}[2]
# both point to the same underlying anonymous hash table

# print the whole thing
foreach $family ( keys %TV ) {
    print "the $family";
    print " is on during @{ $TV{$family}{nights} }\n";
    print "its members are:\n";
    for $who ( @{ $TV{$family}{members} } ) {
        print " $who->{name} ($who->{role}), age $who->{age}\n";
    }
    print "it turns out that $TV{$family}{lead} has ";
    print scalar ( @{ $TV{$family}{kids} } ), " kids named ";
    print join (", ", map { $_->{name} } @{ $TV{$family}{kids} } );
    print "\n";
}
```

**Database Ties**

You cannot easily tie a multilevel data structure (such as a hash of hashes) to a dbm file. The first problem is that all but GDBM and Berkeley DB have size limitations, but beyond that, you also have problems with how references are to be represented on disk. One experimental module that does partially attempt to address this need is the MLDBM module. Check your nearest CPAN site as described in the *perlmodlib* manpage for source code to MLDBM.

**SEE ALSO**

*perlref*(1), *perllol*(1), *perldata*(1), *perlobj*(1)

**AUTHOR**

Tom Christiansen <*tchrist@perl.com*>

Last update: Wed Oct 23 04:57:50 MET DST 1996

## NAME

perlLoL – Manipulating Lists of Lists in Perl

## DESCRIPTION

### Declaration and Access of Lists of Lists

The simplest thing to build is a list of lists (sometimes called an array of arrays). It's reasonably easy to understand, and almost everything that applies here will also be applicable later on with the fancier data structures.

A list of lists, or an array of an array if you would, is just a regular old array `@LoL` that you can get at with two subscripts, like `$LoL[3][2]`. Here's a declaration of the array:

```
    # assign to our array a list of list references
    @LoL = (
           [ "fred", "barney" ],
           [ "george", "jane", "elroy" ],
           [ "homer", "marge", "bart" ],
    );

    print $LoL[2][2];
  bart
```

Now you should be very careful that the outer bracket type is a round one, that is, a parenthesis. That's because you're assigning to an `@list`, so you need parentheses. If you wanted there *not* to be an `@LoL`, but rather just a reference to it, you could do something more like this:

```
    # assign a reference to list of list references
    $ref_to_LoL = [
        [ "fred", "barney", "pebbles", "bambam", "dino", ],
        [ "homer", "bart", "marge", "maggie", ],
        [ "george", "jane", "alroy", "judy", ],
    ];

    print $ref_to_LoL->[2][2];
```

Notice that the outer bracket type has changed, and so our access syntax has also changed. That's because unlike C, in perl you can't freely interchange arrays and references thereto. `$ref_to_LoL` is a reference to an array, whereas `@LoL` is an array proper. Likewise, `$LoL[2]` is not an array, but an array ref. So how come you can write these:

```
    $LoL[2][2]
    $ref_to_LoL->[2][2]
```

instead of having to write these:

```
    $LoL[2]->[2]
    $ref_to_LoL->[2]->[2]
```

Well, that's because the rule is that on adjacent brackets only (whether square or curly), you are free to omit the pointer dereferencing arrow. But you cannot do so for the very first one if it's a scalar containing a reference, which means that `$ref_to_LoL` always needs it.

### Growing Your Own

That's all well and good for declaration of a fixed data structure, but what if you wanted to add new elements on the fly, or build it up entirely from scratch?

First, let's look at reading it in from a file. This is something like adding a row at a time. We'll assume that there's a flat file in which each line is a row and each word an element. If you're trying to develop an `@LoL` list containing all these, here's the right way to do that:

```
while (<>) {
    @tmp = split;
    push @LoL, [ @tmp ];
}
```

You might also have loaded that from a function:

```
for $i ( 1 .. 10 ) {
    $LoL[$i] = [ somefunc($i) ];
}
```

Or you might have had a temporary variable sitting around with the list in it.

```
for $i ( 1 .. 10 ) {
    @tmp = somefunc($i);
    $LoL[$i] = [ @tmp ];
}
```

It's very important that you make sure to use the `[]` list reference constructor. That's because this will be very wrong:

```
$LoL[$i] = @tmp;
```

You see, assigning a named list like that to a scalar just counts the number of elements in `@tmp`, which probably isn't what you want.

If you are running under `use strict`, you'll have to add some declarations to make it happy:

```
use strict;
my(@LoL, @tmp);
while (<>) {
    @tmp = split;
    push @LoL, [ @tmp ];
}
```

Of course, you don't need the temporary array to have a name at all:

```
while (<>) {
    push @LoL, [ split ];
}
```

You also don't have to use *push()*. You could just make a direct assignment if you knew where you wanted to put it:

```
my (@LoL, $i, $line);
for $i ( 0 .. 10 ) {
    $line = <>;
    $LoL[$i] = [ split ' ', $line ];
}
```

or even just

```
my (@LoL, $i);
for $i ( 0 .. 10 ) {
    $LoL[$i] = [ split ' ', <> ];
}
```

You should in general be leery of using potential list functions in a scalar context without explicitly stating such. This would be clearer to the casual reader:

```
my (@LoL, $i);
for $i ( 0 .. 10 ) {
    $LoL[$i] = [ split ' ', scalar(<>) ];
}
```

If you wanted to have a `$ref_to_LoL` variable as a reference to an array, you'd have to do something like this:

```
while (<>) {
    push @$ref_to_LoL, [ split ];
}
```

Now you can add new rows. What about adding new columns? If you're dealing with just matrices, it's often easiest to use simple assignment:

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        $LoL[$x][$y] = func($x, $y);
    }
}

for $x ( 3, 7, 9 ) {
    $LoL[$x][20] += func2($x);
}
```

It doesn't matter whether those elements are already there or not: it'll gladly create them for you, setting intervening elements to `undef` as need be.

If you wanted just to append to a row, you'd have to do something a bit funnier looking:

```
# add new columns to an existing row
push @{ $LoL[0] }, "wilma", "betty";
```

Notice that I *couldn't* say just:

```
push $LoL[0], "wilma", "betty";  # WRONG!
```

In fact, that wouldn't even compile. How come? Because the argument to *push()* must be a real array, not just a reference to such.

**Access and Printing**

Now it's time to print your data structure out. How are you going to do that? Well, if you want only one of the elements, it's trivial:

```
print $LoL[0][0];
```

If you want to print the whole thing, though, you can't say

```
print @LoL;         # WRONG
```

because you'll get just references listed, and perl will never automatically dereference things for you. Instead, you have to roll yourself a loop or two. This prints the whole structure, using the shell-style *for()* construct to loop across the outer set of subscripts.

```
for $aref ( @LoL ) {
    print "\t [ @$aref ],\n";
}
```

If you wanted to keep track of subscripts, you might do this:

```
for $i ( 0 .. $#LoL ) {
    print "\t elt $i is [ @{$LoL[$i]} ],\n";
}
```

or maybe even this. Notice the inner loop.

```
for $i ( 0 .. $#LoL ) {
    for $j ( 0 .. $#{$LoL[$i]} ) {
        print "elt $i $j is $LoL[$i][$j]\n";
    }
}
```

As you can see, it's getting a bit complicated. That's why sometimes is easier to take a temporary on your way through:

```
for $i ( 0 .. $#LoL ) {
    $aref = $LoL[$i];
    for $j ( 0 .. $#{$aref} ) {
        print "elt $i $j is $LoL[$i][$j]\n";
    }
}
```

Hmm... that's still a bit ugly. How about this:

```
for $i ( 0 .. $#LoL ) {
    $aref = $LoL[$i];
    $n = @$aref - 1;
    for $j ( 0 .. $n ) {
        print "elt $i $j is $LoL[$i][$j]\n";
    }
}
```

**Slices**

If you want to get at a slice (part of a row) in a multidimensional array, you're going to have to do some fancy subscripting. That's because while we have a nice synonym for single elements via the pointer arrow for dereferencing, no such convenience exists for slices. (Remember, of course, that you can always write a loop to do a slice operation.)

Here's how to do one operation using a loop. We'll assume an `@LoL` variable as before.

```
@part = ();
$x = 4;
for ($y = 7; $y < 13; $y++) {
    push @part, $LoL[$x][$y];
}
```

That same loop could be replaced with a slice operation:

```
@part = @{ $LoL[4] } [ 7..12 ];
```

but as you might well imagine, this is pretty rough on the reader.

Ah, but what if you wanted a *two-dimensional slice*, such as having `$x` run from 4..8 and `$y` run from 7 to 12? Hmm... here's the simple way:

```
@newLoL = ();
for ($startx = $x = 4; $x <= 8; $x++) {
    for ($starty = $y = 7; $y <= 12; $y++) {
        $newLoL[$x - $startx][$y - $starty] = $LoL[$x][$y];
    }
}
```

We can reduce some of the looping through slices

```
for ($x = 4; $x <= 8; $x++) {
    push @newLoL, [ @{ $LoL[$x] } [ 7..12 ] ];
}
```

If you were into Schwartzian Transforms, you would probably have selected map for that

```
@newLoL = map { [ @{ $LoL[$_] } [ 7..12 ] ] } 4 .. 8;
```

Although if your manager accused of seeking job security (or rapid insecurity) through inscrutable code, it would be hard to argue. :–) If I were you, I'd put that in a function:

```
@newLoL = splice_2D( \@LoL, 4 => 8, 7 => 12 );
sub splice_2D {
    my $lrr = shift;        # ref to list of list refs!
    my ($x_lo, $x_hi,
        $y_lo, $y_hi) = @_;

    return map {
        [ @{ $lrr->[$_] } [ $y_lo .. $y_hi ] ]
    } $x_lo .. $x_hi;
}
```

## SEE ALSO

*perldata* (1), *perlref* (1), *perldsc* (1)

## AUTHOR

Tom Christiansen <*tchrist@perl.com*>

Last update: Thu Jun 4 16:16:23 MDT 1998

## NAME

perltoot − Tom's object-oriented tutorial for perl

## DESCRIPTION

Object-oriented programming is a big seller these days. Some managers would rather have objects than sliced bread. Why is that? What's so special about an object? Just what *is* an object anyway?

An object is nothing but a way of tucking away complex behaviours into a neat little easy-to-use bundle. (This is what professors call abstraction.) Smart people who have nothing to do but sit around for weeks on end figuring out really hard problems make these nifty objects that even regular people can use. (This is what professors call software reuse.) Users (well, programmers) can play with this little bundle all they want, but they aren't to open it up and mess with the insides. Just like an expensive piece of hardware, the contract says that you void the warranty if you muck with the cover. So don't do that.

The heart of objects is the class, a protected little private namespace full of data and functions. A class is a set of related routines that addresses some problem area. You can think of it as a user-defined type. The Perl package mechanism, also used for more traditional modules, is used for class modules as well. Objects "live" in a class, meaning that they belong to some package.

More often than not, the class provides the user with little bundles. These bundles are objects. They know whose class they belong to, and how to behave. Users ask the class to do something, like "give me an object." Or they can ask one of these objects to do something. Asking a class to do something for you is calling a *class method*. Asking an object to do something for you is calling an *object method*. Asking either a class (usually) or an object (sometimes) to give you back an object is calling a *constructor*, which is just a kind of method.

That's all well and good, but how is an object different from any other Perl data type? Just what is an object *really*; that is, what's its fundamental type? The answer to the first question is easy. An object is different from any other data type in Perl in one and only one way: you may dereference it using not merely string or numeric subscripts as with simple arrays and hashes, but with named subroutine calls. In a word, with *methods*.

The answer to the second question is that it's a reference, and not just any reference, mind you, but one whose referent has been *bless*()ed into a particular class (read: package). What kind of reference? Well, the answer to that one is a bit less concrete. That's because in Perl the designer of the class can employ any sort of reference they'd like as the underlying intrinsic data type. It could be a scalar, an array, or a hash reference. It could even be a code reference. But because of its inherent flexibility, an object is usually a hash reference.

### Creating a Class

Before you create a class, you need to decide what to name it. That's because the class (package) name governs the name of the file used to house it, just as with regular modules. Then, that class (package) should provide one or more ways to generate objects. Finally, it should provide mechanisms to allow users of its objects to indirectly manipulate these objects from a distance.

For example, let's make a simple Person class module. It gets stored in the file Person.pm. If it were called a Happy::Person class, it would be stored in the file Happy/Person.pm, and its package would become Happy::Person instead of just Person. (On a personal computer not running Unix or Plan 9, but something like MacOS or VMS, the directory separator may be different, but the principle is the same.) Do not assume any formal relationship between modules based on their directory names. This is merely a grouping convenience, and has no effect on inheritance, variable accessibility, or anything else.

For this module we aren't going to use Exporter, because we're a well-behaved class module that doesn't export anything at all. In order to manufacture objects, a class needs to have a *constructor method*. A constructor gives you back not just a regular data type, but a brand-new object in that class. This magic is taken care of by the *bless()* function, whose sole purpose is to enable its referent to be used as an object. Remember: being an object really means nothing more than that methods may now be called against it.

While a constructor may be named anything you'd like, most Perl programmers seem to like to call theirs *new()*. However, *new()* is not a reserved word, and a class is under no obligation to supply such. Some programmers have also been known to use a function with the same name as the class as the constructor.

### Object Representation

By far the most common mechanism used in Perl to represent a Pascal record, a C struct, or a C++ class is an anonymous hash. That's because a hash has an arbitrary number of data fields, each conveniently accessed by an arbitrary name of your own devising.

If you were just doing a simple struct-like emulation, you would likely go about it something like this:

```
$rec = {
    name  => "Jason",
    age   => 23,
    peers => [ "Norbert", "Rhys", "Phineas"],
};
```

If you felt like it, you could add a bit of visual distinction by up-casing the hash keys:

```
$rec = {
    NAME  => "Jason",
    AGE   => 23,
    PEERS => [ "Norbert", "Rhys", "Phineas"],
};
```

And so you could get at `$rec->{NAME}` to find "Jason", or `@{ $rec->{PEERS} }` to get at "Norbert", "Rhys", and "Phineas". (Have you ever noticed how many 23-year-old programmers seem to be named "Jason" these days? :-)

This same model is often used for classes, although it is not considered the pinnacle of programming propriety for folks from outside the class to come waltzing into an object, brazenly accessing its data members directly. Generally speaking, an object should be considered an opaque cookie that you use *object methods* to access. Visually, methods look like you're dereffing a reference using a function name instead of brackets or braces.

### Class Interface

Some languages provide a formal syntactic interface to a class's methods, but Perl does not. It relies on you to read the documentation of each class. If you try to call an undefined method on an object, Perl won't complain, but the program will trigger an exception while it's running. Likewise, if you call a method expecting a prime number as its argument with a non-prime one instead, you can't expect the compiler to catch this. (Well, you can expect it all you like, but it's not going to happen.)

Let's suppose you have a well-educated user of your Person class, someone who has read the docs that explain the prescribed interface. Here's how they might use the Person class:

```
use Person;

$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( "Norbert", "Rhys", "Phineas" );

push @All_Recs, $him;  # save object in array for later

printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(", ", $him->peers), "\n";

printf "Last rec's name is %s\n", $All_Recs[-1]->name;
```

As you can see, the user of the class doesn't know (or at least, has no business paying attention to the fact) that the object has one particular implementation or another. The interface to the class and its objects is exclusively via methods, and that's all the user of the class should ever play with.

**Constructors and Instance Methods**

Still, *someone* has to know what's in the object. And that someone is the class. It implements methods that the programmer uses to access the object. Here's how to implement the Person class using the standard hash-ref-as-an-object idiom. We'll make a class method called *new()* to act as the constructor, and three object methods called *name()*, *age()*, and *peers()* to get at per-object data hidden away in our anonymous hash.

```
package Person;
use strict;

##################################################
## the object constructor (simplistic version)  ##
##################################################
sub new {
    my $self  = {};
    $self->{NAME}   = undef;
    $self->{AGE}    = undef;
    $self->{PEERS}  = [];
    bless($self);           # but see below
    return $self;
}

##############################################
## methods to access per-object data        ##
##                                          ##
## With args, they set the value.  Without  ##
## any, they only retrieve it/them.         ##
##############################################

sub name {
    my $self = shift;
    if (@_) { $self->{NAME} = shift }
    return $self->{NAME};
}

sub age {
    my $self = shift;
    if (@_) { $self->{AGE} = shift }
    return $self->{AGE};
}

sub peers {
    my $self = shift;
    if (@_) { @{ $self->{PEERS} } = @_ }
    return @{ $self->{PEERS} };
}

1;  # so the require or use succeeds
```

We've created three methods to access an object's data, *name()*, *age()*, and *peers()*. These are all substantially similar. If called with an argument, they set the appropriate field; otherwise they return the value held by that field, meaning the value of that hash key.

### Planning for the Future: Better Constructors

Even though at this point you may not even know what it means, someday you're going to worry about inheritance. (You can safely ignore this for now and worry about it later if you'd like.) To ensure that this all works out smoothly, you must use the double-argument form of *bless()*. The second argument is the class into which the referent will be blessed. By not assuming our own class as the default second argument and instead using the class passed into us, we make our constructor inheritable.

While we're at it, let's make our constructor a bit more flexible. Rather than being uniquely a class method, we'll set it up so that it can be called as either a class method *or* an object method. That way you can say:

```
$me  = Person->new();
$him = $me->new();
```

To do this, all we have to do is check whether what was passed in was a reference or not. If so, we were invoked as an object method, and we need to extract the package (class) using the *ref()* function. If not, we just use the string passed in as the package name for blessing our referent.

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{NAME}   = undef;
    $self->{AGE}    = undef;
    $self->{PEERS}  = [];
    bless ($self, $class);
    return $self;
}
```

That's about all there is for constructors. These methods bring objects to life, returning neat little opaque bundles to the user to be used in subsequent method calls.

### Destructors

Every story has a beginning and an end. The beginning of the object's story is its constructor, explicitly called when the object comes into existence. But the ending of its story is the *destructor*, a method implicitly called when an object leaves this life. Any per-object clean-up code is placed in the destructor, which must (in Perl) be called DESTROY.

If constructors can have arbitrary names, then why not destructors? Because while a constructor is explicitly called, a destructor is not. Destruction happens automatically via Perl's garbage collection (GC) system, which is a quick but somewhat lazy reference-based GC system. To know what to call, Perl insists that the destructor be named DESTROY. Perl's notion of the right time to call a destructor is not well-defined currently, which is why your destructors should not rely on when they are called.

Why is DESTROY in all caps? Perl on occasion uses purely uppercase function names as a convention to indicate that the function will be automatically called by Perl in some way. Others that are called implicitly include BEGIN, END, AUTOLOAD, plus all methods used by tied objects, described in the *perltie* manpage.

In really good object-oriented programming languages, the user doesn't care when the destructor is called. It just happens when it's supposed to. In low-level languages without any GC at all, there's no way to depend on this happening at the right time, so the programmer must explicitly call the destructor to clean up memory and state, crossing their fingers that it's the right time to do so. Unlike C++, an object destructor is nearly never needed in Perl, and even when it is, explicit invocation is uncalled for. In the case of our Person class, we don't need a destructor because Perl takes care of simple matters like memory deallocation.

The only situation where Perl's reference-based GC won't work is when there's a circularity in the data structure, such as:

```
$this->{WHATEVER} = $this;
```

In that case, you must delete the self-reference manually if you expect your program not to leak memory. While admittedly error-prone, this is the best we can do right now. Nonetheless, rest assured that when your program is finished, its objects' destructors are all duly called. So you are guaranteed that an object *eventually* gets properly destroyed, except in the unique case of a program that never exits. (If you're running Perl embedded in another application, this full GC pass happens a bit more frequently—whenever a thread shuts down.)

### Other Object Methods

The methods we've talked about so far have either been constructors or else simple "data methods", interfaces to data stored in the object. These are a bit like an object's data members in the C++ world, except that strangers don't access them as data. Instead, they should only access the object's data indirectly via its methods. This is an important rule: in Perl, access to an object's data should *only* be made through methods.

Perl doesn't impose restrictions on who gets to use which methods. The public-versus-private distinction is by convention, not syntax. (Well, unless you use the Alias module described below in the section on *Data Members as Variables*.) Occasionally you'll see method names beginning or ending with an underscore or two. This marking is a convention indicating that the methods are private to that class alone and sometimes to its closest acquaintances, its immediate subclasses. But this distinction is not enforced by Perl itself. It's up to the programmer to behave.

There's no reason to limit methods to those that simply access data. Methods can do anything at all. The key point is that they're invoked against an object or a class. Let's say we'd like object methods that do more than fetch or set one particular field.

```
sub exclaim {
    my $self = shift;
    return sprintf "Hi, I'm %s, age %d, working with %s",
        $self->{NAME}, $self->{AGE}, join(", ", $self->{PEERS});
}
```

Or maybe even one like this:

```
sub happy_birthday {
    my $self = shift;
    return ++$self->{AGE};
}
```

Some might argue that one should go at these this way:

```
sub exclaim {
    my $self = shift;
    return sprintf "Hi, I'm %s, age %d, working with %s",
        $self->name, $self->age, join(", ", $self->peers);
}

sub happy_birthday {
    my $self = shift;
    return $self->age( $self->age() + 1 );
}
```

But since these methods are all executing in the class itself, this may not be critical. There are tradeoffs to be made. Using direct hash access is faster (about an order of magnitude faster, in fact), and it's more convenient when you want to interpolate in strings. But using methods (the external interface) internally shields not just the users of your class but even you yourself from changes in your data representation.

**Class Data**

What about "class data", data items common to each object in a class? What would you want that for? Well, in your Person class, you might like to keep track of the total people alive. How do you implement that?

You *could* make it a global variable called `$Person::Census`. But about only reason you'd do that would be if you *wanted* people to be able to get at your class data directly. They could just say `$Person::Census` and play around with it. Maybe this is ok in your design scheme. You might even conceivably want to make it an exported variable. To be exportable, a variable must be a (package) global. If this were a traditional module rather than an object-oriented one, you might do that.

While this approach is expected in most traditional modules, it's generally considered rather poor form in most object modules. In an object module, you should set up a protective veil to separate interface from implementation. So provide a class method to access class data just as you provide object methods to access object data.

So, you *could* still keep `$Census` as a package global and rely upon others to honor the contract of the module and therefore not play around with its implementation. You could even be supertricky and make `$Census` a tied object as described in the *perltie* manpage, thereby intercepting all accesses.

But more often than not, you just want to make your class data a file-scoped lexical. To do so, simply put this at the top of the file:

```
my $Census = 0;
```

Even though the scope of a *my()* normally expires when the block in which it was declared is done (in this case the whole file being required or used), Perl's deep binding of lexical variables guarantees that the variable will not be deallocated, remaining accessible to functions declared within that scope. This doesn't work with global variables given temporary values via *local()*, though.

Irrespective of whether you leave `$Census` a package global or make it instead a file-scoped lexical, you should make these changes to your *Person::new()* constructor:

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $Census++;
    $self->{NAME}   = undef;
    $self->{AGE}    = undef;
    $self->{PEERS}  = [];
    bless ($self, $class);
    return $self;
}

sub population {
    return $Census;
}
```

Now that we've done this, we certainly do need a destructor so that when Person is destroyed, the `$Census` goes down. Here's how this could be done:

```
sub DESTROY { --$Census }
```

Notice how there's no memory to deallocate in the destructor? That's something that Perl takes care of for you all by itself.

### Accessing Class Data

It turns out that this is not really a good way to go about handling class data. A good scalable rule is that *you must never reference class data directly from an object method*. Otherwise you aren't building a scalable, inheritable class. The object must be the rendezvous point for all operations, especially from an object method. The globals (class data) would in some sense be in the ‘‘wrong'' package in your derived classes. In Perl, methods execute in the context of the class they were defined in, *not* that of the object that triggered them. Therefore, namespace visibility of package globals in methods is unrelated to inheritance.

Got that? Maybe not. Ok, let's say that some other class ‘‘borrowed'' (well, inherited) the DESTROY method as it was defined above. When those objects are destroyed, the original `$Census` variable will be altered, not the one in the new class's package namespace. Perhaps this is what you want, but probably it isn't.

Here's how to fix this. We'll store a reference to the data in the value accessed by the hash key ‘‘_CENSUS''. Why the underscore? Well, mostly because an initial underscore already conveys strong feelings of magicalness to a C programmer. It's really just a mnemonic device to remind ourselves that this field is special and not to be used as a public data member in the same way that NAME, AGE, and PEERS are. (Because we've been developing this code under the strict pragma, prior to perl version 5.004 we'll have to quote the field name.)

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{NAME}     = undef;
    $self->{AGE}      = undef;
    $self->{PEERS}    = [];
    # "private" data
    $self->{"_CENSUS"} = \$Census;
    bless ($self, $class);
    ++ ${ $self->{"_CENSUS"} };
    return $self;
}

sub population {
    my $self = shift;
    if (ref $self) {
        return ${ $self->{"_CENSUS"} };
    } else {
        return $Census;
    }
}

sub DESTROY {
    my $self = shift;
    -- ${ $self->{"_CENSUS"} };
}
```

### Debugging Methods

It's common for a class to have a debugging mechanism. For example, you might want to see when objects are created or destroyed. To do that, add a debugging variable as a file-scoped lexical. For this, we'll pull in the standard Carp module to emit our warnings and fatal messages. That way messages will come out with the caller's filename and line number instead of our own; if we wanted them to be from our own perspective, we'd just use *die()* and *warn()* directly instead of *croak()* and *carp()* respectively.

```
use Carp;
my $Debugging = 0;
```

Now add a new class method to access the variable.

```
sub debug {
    my $class = shift;
    if (ref $class)  { confess "Class method called as object method" }
    unless (@_ == 1) { confess "usage: CLASSNAME->debug(level)" }
    $Debugging = shift;
}
```

Now fix up DESTROY to murmur a bit as the moribund object expires:

```
sub DESTROY {
    my $self = shift;
    if ($Debugging) { carp "Destroying $self " . $self->name }
    -- ${ $self->{"_CENSUS"} };
}
```

One could conceivably make a per-object debug state. That way you could call both of these:

```
Person->debug(1);   # entire class
$him->debug(1);     # just this object
```

To do so, we need our debugging method to be a "bimodal" one, one that works on both classes *and* objects. Therefore, adjust the *debug()* and DESTROY methods as follows:

```
sub debug {
    my $self = shift;
    confess "usage: thing->debug(level)"    unless @_ == 1;
    my $level = shift;
    if (ref($self))  {
        $self->{"_DEBUG"} = $level;         # just myself
    } else {
        $Debugging        = $level;         # whole class
    }
}

sub DESTROY {
    my $self = shift;
    if ($Debugging || $self->{"_DEBUG"}) {
        carp "Destroying $self " . $self->name;
    }
    -- ${ $self->{"_CENSUS"} };
}
```

What happens if a derived class (which we'll call Employee) inherits methods from this Person base class? Then `Employee->debug()`, when called as a class method, manipulates `$Person::Debugging` not `$Employee::Debugging`.

### Class Destructors

The object destructor handles the death of each distinct object. But sometimes you want a bit of cleanup when the entire class is shut down, which currently only happens when the program exits. To make such a *class destructor*, create a function in that class's package named END. This works just like the END function in traditional modules, meaning that it gets called whenever your program exits unless it execs or dies of an uncaught signal. For example,

```
    sub END {
        if ($Debugging) {
            print "All persons are going away now.\n";
        }
    }
```

When the program exits, all the class destructors (END functions) are be called in the opposite order that they were loaded in (LIFO order).

### Documenting the Interface

And there you have it: we've just shown you the *implementation* of this Person class. Its *interface* would be its documentation. Usually this means putting it in pod ("plain old documentation") format right there in the same file. In our Person example, we would place the following docs anywhere in the Person.pm file. Even though it looks mostly like code, it's not. It's embedded documentation such as would be used by the pod2man, pod2html, or pod2text programs. The Perl compiler ignores pods entirely, just as the translators ignore code. Here's an example of some pods describing the informal interface:

```
    =head1 NAME

    Person - class to implement people

    =head1 SYNOPSIS

     use Person;

     #################
     # class methods #
     #################
     $ob    = Person->new;
     $count = Person->population;

     #######################
     # object data methods #
     #######################

     ### get versions ###
         $who   = $ob->name;
         $years = $ob->age;
         @pals  = $ob->peers;

     ### set versions ###
         $ob->name("Jason");
         $ob->age(23);
         $ob->peers( "Norbert", "Rhys", "Phineas" );

     ########################
     # other object methods #
     ########################

     $phrase = $ob->exclaim;
     $ob->happy_birthday;

    =head1 DESCRIPTION
```

```
The Person class implements dah dee dah dee dah....
```

That's all there is to the matter of interface versus implementation. A programmer who opens up the module and plays around with all the private little shiny bits that were safely locked up behind the interface contract has voided the warranty, and you shouldn't worry about their fate.

**Aggregation**

Suppose you later want to change the class to implement better names. Perhaps you'd like to support both given names (called Christian names, irrespective of one's religion) and family names (called surnames), plus nicknames and titles. If users of your Person class have been properly accessing it through its documented interface, then you can easily change the underlying implementation. If they haven't, then they lose and it's their fault for breaking the contract and voiding their warranty.

To do this, we'll make another class, this one called Fullname. What's the Fullname class look like? To answer that question, you have to first figure out how you want to use it. How about we use it this way:

```
$him = Person->new();
$him->fullname->title("St");
$him->fullname->christian("Thomas");
$him->fullname->surname("Aquinas");
$him->fullname->nickname("Tommy");
printf "His normal name is %s\n", $him->name;
printf "But his real name is %s\n", $him->fullname->as_string;
```

Ok. To do this, we'll change *Person::new()* so that it supports a full name field this way:

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {};
    $self->{FULLNAME} = Fullname->new();
    $self->{AGE}      = undef;
    $self->{PEERS}    = [];
    $self->{"_CENSUS"} = \$Census;
    bless ($self, $class);
    ++ ${ $self->{"_CENSUS"} };
    return $self;
}

sub fullname {
    my $self = shift;
    return $self->{FULLNAME};
}
```

Then to support old code, define *Person::name()* this way:

```
sub name {
    my $self = shift;
    return $self->{FULLNAME}->nickname(@_)
      ||   $self->{FULLNAME}->christian(@_);
}
```

Here's the Fullname class. We'll use the same technique of using a hash reference to hold data fields, and methods by the appropriate name to access them:

```
package Fullname;
use strict;
```

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = {
        TITLE       => undef,
        CHRISTIAN   => undef,
        SURNAME     => undef,
        NICK        => undef,
    };
    bless ($self, $class);
    return $self;
}

sub christian {
    my $self = shift;
    if (@_) { $self->{CHRISTIAN} = shift }
    return $self->{CHRISTIAN};
}

sub surname {
    my $self = shift;
    if (@_) { $self->{SURNAME} = shift }
    return $self->{SURNAME};
}

sub nickname {
    my $self = shift;
    if (@_) { $self->{NICK} = shift }
    return $self->{NICK};
}

sub title {
    my $self = shift;
    if (@_) { $self->{TITLE} = shift }
    return $self->{TITLE};
}

sub as_string {
    my $self = shift;
    my $name = join(" ", @$self{'CHRISTIAN', 'SURNAME'});
    if ($self->{TITLE}) {
        $name = $self->{TITLE} . " " . $name;
    }
    return $name;
}

1;
```

Finally, here's the test program:

```
#!/usr/bin/perl -w
use strict;
use Person;
sub END { show_census() }
```

```
sub show_census ()  {
    printf "Current population: %d\n", Person->population;
}

Person->debug(1);

show_census();

my $him = Person->new();

$him->fullname->christian("Thomas");
$him->fullname->surname("Aquinas");
$him->fullname->nickname("Tommy");
$him->fullname->title("St");
$him->age(1);

printf "%s is really %s.\n", $him->name, $him->fullname;
printf "%s's age: %d.\n", $him->name, $him->age;
$him->happy_birthday;
printf "%s's age: %d.\n", $him->name, $him->age;

show_census();
```

**Inheritance**

Object-oriented programming systems all support some notion of inheritance. Inheritance means allowing one class to piggy-back on top of another one so you don't have to write the same code again and again. It's about software reuse, and therefore related to Laziness, the principal virtue of a programmer. (The import/export mechanisms in traditional modules are also a form of code reuse, but a simpler one than the true inheritance that you find in object modules.)

Sometimes the syntax of inheritance is built into the core of the language, and sometimes it's not. Perl has no special syntax for specifying the class (or classes) to inherit from. Instead, it's all strictly in the semantics. Each package can have a variable called `@ISA`, which governs (method) inheritance. If you try to call a method on an object or class, and that method is not found in that object's package, Perl then looks to `@ISA` for other packages to go looking through in search of the missing method.

Like the special per-package variables recognized by Exporter (such as `@EXPORT`, `@EXPORT_OK`, `@EXPORT_FAIL`, `%EXPORT_TAGS`, and `$VERSION`), the `@ISA` array *must* be a package-scoped global and not a file-scoped lexical created via *my()*. Most classes have just one item in their `@ISA` array. In this case, we have what's called ‘‘single inheritance'', or SI for short.

Consider this class:

```
package Employee;
use Person;
@ISA = ("Person");
1;
```

Not a lot to it, eh? All it's doing so far is loading in another class and stating that this one will inherit methods from that other class if need be. We have given it none of its own methods. We rely upon an Employee to behave just like a Person.

Setting up an empty class like this is called the ‘‘empty subclass test''; that is, making a derived class that does nothing but inherit from a base class. If the original base class has been designed properly, then the new derived class can be used as a drop-in replacement for the old one. This means you should be able to write a program like this:

```
use Employee;
my $empl = Employee->new();
$empl->name("Jason");
$empl->age(23);
printf "%s is age %d.\n", $empl->name, $empl->age;
```

By proper design, we mean always using the two-argument form of *bless()*, avoiding direct access of global data, and not exporting anything. If you look back at the *Person::new()* function we defined above, we were careful to do that. There's a bit of package data used in the constructor, but the reference to this is stored on the object itself and all other methods access package data via that reference, so we should be ok.

What do we mean by the *Person::new()* function -- isn't that actually a method? Well, in principle, yes. A method is just a function that expects as its first argument a class name (package) or object (blessed reference). *Person::new()* is the function that both the `Person->new()` method and the `Employee->new()` method end up calling. Understand that while a method call looks a lot like a function call, they aren't really quite the same, and if you treat them as the same, you'll very soon be left with nothing but broken programs. First, the actual underlying calling conventions are different: method calls get an extra argument. Second, function calls don't do inheritance, but methods do.

```
    Method Call             Resulting Function Call
    -----------             ------------------------
    Person->new()           Person::new("Person")
    Employee->new()         Person::new("Employee")
```

So don't use function calls when you mean to call a method.

If an employee is just a Person, that's not all too very interesting. So let's add some other methods. We'll give our employee data fields to access their salary, their employee ID, and their start date.

If you're getting a little tired of creating all these nearly identical methods just to get at the object's data, do not despair. Later, we'll describe several different convenience mechanisms for shortening this up. Meanwhile, here's the straight-forward way:

```
sub salary {
    my $self = shift;
    if (@_) { $self->{SALARY} = shift }
    return $self->{SALARY};
}

sub id_number {
    my $self = shift;
    if (@_) { $self->{ID} = shift }
    return $self->{ID};
}

sub start_date {
    my $self = shift;
    if (@_) { $self->{START_DATE} = shift }
    return $self->{START_DATE};
}
```

**Overridden Methods**

What happens when both a derived class and its base class have the same method defined? Well, then you get the derived class's version of that method. For example, let's say that we want the *peers()* method called on an employee to act a bit differently. Instead of just returning the list of peer names, let's return slightly different strings. So doing this:

```
$empl->peers("Peter", "Paul", "Mary");
printf "His peers are: %s\n", join(", ", $empl->peers);
```

will produce:

```
His peers are: PEON=PETER, PEON=PAUL, PEON=MARY
```

To do this, merely add this definition into the Employee.pm file:

```
sub peers {
    my $self = shift;
    if (@_) { @{ $self->{PEERS} } = @_ }
    return map { "PEON=\U$_" } @{ $self->{PEERS} };
}
```

There, we've just demonstrated the high-falutin' concept known in certain circles as *polymorphism*. We've taken on the form and behaviour of an existing object, and then we've altered it to suit our own purposes. This is a form of Laziness. (Getting polymorphed is also what happens when the wizard decides you'd look better as a frog.)

Every now and then you'll want to have a method call trigger both its derived class (also known as "subclass") version as well as its base class (also known as "superclass") version. In practice, constructors and destructors are likely to want to do this, and it probably also makes sense in the *debug()* method we showed previously.

To do this, add this to Employee.pm:

```
use Carp;
my $Debugging = 0;

sub debug {
    my $self = shift;
    confess "usage: thing->debug(level)"    unless @_ == 1;
    my $level = shift;
    if (ref($self))  {
        $self->{"_DEBUG"} = $level;
    } else {
        $Debugging = $level;            # whole class
    }
    Person::debug($self, $Debugging);   # don't really do this
}
```

As you see, we turn around and call the Person package's *debug()* function. But this is far too fragile for good design. What if Person doesn't have a *debug()* function, but is inheriting *its debug()* method from elsewhere? It would have been slightly better to say

```
Person->debug($Debugging);
```

But even that's got too much hard-coded. It's somewhat better to say

```
$self->Person::debug($Debugging);
```

Which is a funny way to say to start looking for a *debug()* method up in Person. This strategy is more often seen on overridden object methods than on overridden class methods.

There is still something a bit off here. We've hard-coded our superclass's name. This in particular is bad if you change which classes you inherit from, or add others. Fortunately, the pseudoclass SUPER comes to the rescue here.

```
$self->SUPER::debug($Debugging);
```

This way it starts looking in my class's `@ISA`. This only makes sense from *within* a method call, though. Don't try to access anything in SUPER:: from anywhere else, because it doesn't exist outside an overridden method call.

Things are getting a bit complicated here. Have we done anything we shouldn't? As before, one way to test whether we're designing a decent class is via the empty subclass test. Since we already have an Employee class that we're trying to check, we'd better get a new empty subclass that can derive from Employee. Here's one:

```
package Boss;
use Employee;         # :-)
@ISA = qw(Employee);
```

And here's the test program:

```
#!/usr/bin/perl -w
use strict;
use Boss;
Boss->debug(1);

my $boss = Boss->new();

$boss->fullname->title("Don");
$boss->fullname->surname("Pichon Alvarez");
$boss->fullname->christian("Federico Jesus");
$boss->fullname->nickname("Fred");

$boss->age(47);
$boss->peers("Frank", "Felipe", "Faust");

printf "%s is age %d.\n", $boss->fullname, $boss->age;
printf "His peers are: %s\n", join(", ", $boss->peers);
```

Running it, we see that we're still ok. If you'd like to dump out your object in a nice format, somewhat like the way the 'x' command works in the debugger, you could use the Data::Dumper module from CPAN this way:

```
use Data::Dumper;
print "Here's the boss:\n";
print Dumper($boss);
```

Which shows us something like this:

```
Here's the boss:
$VAR1 = bless( {
     _CENSUS => \1,
     FULLNAME => bless( {
                          TITLE => 'Don',
                          SURNAME => 'Pichon Alvarez',
                          NICK => 'Fred',
                          CHRISTIAN => 'Federico Jesus'
                        }, 'Fullname' ),
     AGE => 47,
     PEERS => [
                'Frank',
                'Felipe',
                'Faust'
              ]
   }, 'Boss' );
```

Hm.... something's missing there. What about the salary, start date, and ID fields? Well, we never set them to anything, even undef, so they don't show up in the hash's keys. The Employee class has no *new()* method of its own, and the *new()* method in Person doesn't know about Employees. (Nor should it: proper OO design dictates that a subclass be allowed to know about its immediate superclass, but never vice-versa.) So let's fix up *Employee::new()* this way:

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self  = $class->SUPER::new();
    $self->{SALARY}        = undef;
    $self->{ID}            = undef;
    $self->{START_DATE}    = undef;
    bless ($self, $class);          # reconsecrate
    return $self;
}
```

Now if you dump out an Employee or Boss object, you'll find that new fields show up there now.

**Multiple Inheritance**

Ok, at the risk of confusing beginners and annoying OO gurus, it's time to confess that Perl's object system includes that controversial notion known as multiple inheritance, or MI for short. All this means is that rather than having just one parent class who in turn might itself have a parent class, etc., that you can directly inherit from two or more parents. It's true that some uses of MI can get you into trouble, although hopefully not quite so much trouble with Perl as with dubiously-OO languages like C++.

The way it works is actually pretty simple: just put more than one package name in your `@ISA` array. When it comes time for Perl to go finding methods for your object, it looks at each of these packages in order. Well, kinda. It's actually a fully recursive, depth-first order. Consider a bunch of `@ISA` arrays like this:

```
@First::ISA    = qw( Alpha );
@Second::ISA   = qw( Beta );
@Third::ISA    = qw( First Second );
```

If you have an object of class Third:

```
my $ob = Third->new();
$ob->spin();
```

How do we find a *spin()* method (or a *new()* method for that matter)? Because the search is depth-first, classes will be looked up in the following order: Third, First, Alpha, Second, and Beta.

In practice, few class modules have been seen that actually make use of MI. One nearly always chooses simple containership of one class within another over MI. That's why our Person object *contained* a Fullname object. That doesn't mean it *was* one.

However, there is one particular area where MI in Perl is rampant: borrowing another class's class methods. This is rather common, especially with some bundled "objectless" classes, like Exporter, DynaLoader, AutoLoader, and SelfLoader. These classes do not provide constructors; they exist only so you may inherit their class methods. (It's not entirely clear why inheritance was done here rather than traditional module importation.)

For example, here is the POSIX module's `@ISA`:

```
package POSIX;
@ISA = qw(Exporter DynaLoader);
```

The POSIX module isn't really an object module, but then, neither are Exporter or DynaLoader. They're just lending their classes' behaviours to POSIX.

Why don't people use MI for object methods much? One reason is that it can have complicated side-effects. For one thing, your inheritance graph (no longer a tree) might converge back to the same base class. Although Perl guards against recursive inheritance, merely having parents who are related to each other via a common ancestor, incestuous though it sounds, is not forbidden. What if in our Third class shown above we wanted its *new()* method to also call both overridden constructors in its two parent classes? The SUPER notation would only find the first one. Also, what about if the Alpha and Beta classes both had a common ancestor, like Nought? If you kept climbing up the inheritance tree calling overridden methods, you'd end up calling *Nought::new()* twice, which might well be a bad idea.

### UNIVERSAL: The Root of All Objects

Wouldn't it be convenient if all objects were rooted at some ultimate base class? That way you could give every object common methods without having to go and add it to each and every `@ISA`. Well, it turns out that you can. You don't see it, but Perl tacitly and irrevocably assumes that there's an extra element at the end of `@ISA`: the class UNIVERSAL. In version 5.003, there were no predefined methods there, but you could put whatever you felt like into it.

However, as of version 5.004 (or some subversive releases, like 5.003_08), UNIVERSAL has some methods in it already. These are builtin to your Perl binary, so they don't take any extra time to load. Predefined methods include *isa()*, *can()*, and *VERSION()*. *isa()* tells you whether an object or class "is" another one without having to traverse the hierarchy yourself:

```
$has_io = $fd->isa("IO::Handle");
$itza_handle = IO::Socket->isa("IO::Handle");
```

The *can()* method, called against that object or class, reports back whether its string argument is a callable method name in that class. In fact, it gives you back a function reference to that method:

```
$his_print_method = $obj->can('as_string');
```

Finally, the VERSION method checks whether the class (or the object's class) has a package global called `$VERSION` that's high enough, as in:

```
Some_Module->VERSION(3.0);
$his_vers = $ob->VERSION();
```

However, we don't usually call VERSION ourselves. (Remember that an all uppercase function name is a Perl convention that indicates that the function will be automatically used by Perl in some way.) In this case, it happens when you say

```
use Some_Module 3.0;
```

If you wanted to add version checking to your Person class explained above, just add this to Person.pm:

```
use vars qw($VERSION);
$VERSION = '1.1';
```

and then in Employee.pm could you can say

```
use Employee 1.1;
```

And it would make sure that you have at least that version number or higher available. This is not the same as loading in that exact version number. No mechanism currently exists for concurrent installation of multiple versions of a module. Lamentably.

**Alternate Object Representations**

Nothing requires objects to be implemented as hash references. An object can be any sort of reference so long as its referent has been suitably blessed. That means scalar, array, and code references are also fair game.

A scalar would work if the object has only one datum to hold. An array would work for most cases, but makes inheritance a bit dodgy because you have to invent new indices for the derived classes.

### Arrays as Objects

If the user of your class honors the contract and sticks to the advertised interface, then you can change its underlying interface if you feel like it. Here's another implementation that conforms to the same interface specification. This time we'll use an array reference instead of a hash reference to represent the object.

```
package Person;
use strict;

my($NAME, $AGE, $PEERS) = ( 0 .. 2 );

############################################
## the object constructor (array version) ##
############################################
sub new {
    my $self = [];
    $self->[$NAME]   = undef;  # this is unnecessary
    $self->[$AGE]    = undef;  # as is this
    $self->[$PEERS]  = [];     # but this isn't, really
    bless($self);
    return $self;
}

sub name {
    my $self = shift;
    if (@_) { $self->[$NAME] = shift }
    return $self->[$NAME];
}

sub age {
    my $self = shift;
    if (@_) { $self->[$AGE] = shift }
    return $self->[$AGE];
}
```

```
sub peers {
    my $self = shift;
    if (@_) { @{ $self->[$PEERS] } = @_ }
    return @{ $self->[$PEERS] };
}

1;  # so the require or use succeeds
```

You might guess that the array access would be a lot faster than the hash access, but they're actually comparable. The array is a *little* bit faster, but not more than ten or fifteen percent, even when you replace the variables above like `$AGE` with literal numbers, like 1. A bigger difference between the two approaches can be found in memory use. A hash representation takes up more memory than an array representation because you have to allocate memory for the keys as well as for the values. However, it really isn't that bad, especially since as of version 5.004, memory is only allocated once for a given hash key, no matter how many hashes have that key. It's expected that sometime in the future, even these differences will fade into obscurity as more efficient underlying representations are devised.

Still, the tiny edge in speed (and somewhat larger one in memory) is enough to make some programmers choose an array representation for simple classes. There's still a little problem with scalability, though, because later in life when you feel like creating subclasses, you'll find that hashes just work out better.

### Closures as Objects

Using a code reference to represent an object offers some fascinating possibilities. We can create a new anonymous function (closure) who alone in all the world can see the object's data. This is because we put the data into an anonymous hash that's lexically visible only to the closure we create, bless, and return as the object. This object's methods turn around and call the closure as a regular subroutine call, passing it the field we want to affect. (Yes, the double-function call is slow, but if you wanted fast, you wouldn't be using objects at all, eh? :–)

Use would be similar to before:

```
use Person;
$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( [ "Norbert", "Rhys", "Phineas" ] );
printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(", ", @{$him->peers}), "\n";
```

but the implementation would be radically, perhaps even sublimely different:

```
package Person;
```

```
sub new {
     my $that  = shift;
     my $class = ref($that) || $that;
     my $self = {
        NAME  => undef,
        AGE   => undef,
        PEERS => [],
     };
     my $closure = sub {
        my $field = shift;
        if (@_) { $self->{$field} = shift }
        return    $self->{$field};
    };
    bless($closure, $class);
    return $closure;
}

sub name   { &{ $_[0] }("NAME",  @_[ 1 .. $#_ ] ) }
sub age    { &{ $_[0] }("AGE",   @_[ 1 .. $#_ ] ) }
sub peers  { &{ $_[0] }("PEERS", @_[ 1 .. $#_ ] ) }

1;
```

Because this object is hidden behind a code reference, it's probably a bit mysterious to those whose background is more firmly rooted in standard procedural or object-based programming languages than in functional programming languages whence closures derive. The object created and returned by the *new()* method is itself not a data reference as we've seen before. It's an anonymous code reference that has within it access to a specific version (lexical binding and instantiation) of the object's data, which are stored in the private variable `$self`. Although this is the same function each time, it contains a different version of `$self`.

When a method like `$him->name("Jason")` is called, its implicit zeroth argument is the invoking object— just as it is with all method calls. But in this case, it's our code reference (something like a function pointer in C++, but with deep binding of lexical variables). There's not a lot to be done with a code reference beyond calling it, so that's just what we do when we say `&{$_[0]}`. This is just a regular function call, not a method call. The initial argument is the string "NAME", and any remaining arguments are whatever had been passed to the method itself.

Once we're executing inside the closure that had been created in *new()*, the `$self` hash reference suddenly becomes visible. The closure grabs its first argument ("NAME" in this case because that's what the *name()* method passed it), and uses that string to subscript into the private hash hidden in its unique version of `$self`.

Nothing under the sun will allow anyone outside the executing method to be able to get at this hidden data. Well, nearly nothing. You *could* single step through the program using the debugger and find out the pieces while you're in the method, but everyone else is out of luck.

There, if that doesn't excite the Scheme folks, then I just don't know what will. Translation of this technique into C++, Java, or any other braindead-static language is left as a futile exercise for aficionados of those camps.

You could even add a bit of nosiness via the *caller()* function and make the closure refuse to operate unless called via its own package. This would no doubt satisfy certain fastidious concerns of programming police and related puritans.

If you were wondering when Hubris, the third principle virtue of a programmer, would come into play, here you have it. (More seriously, Hubris is just the pride in craftsmanship that comes from having written a sound bit of well-designed code.)

**AUTOLOAD: Proxy Methods**

Autoloading is a way to intercept calls to undefined methods. An autoload routine may choose to create a new function on the fly, either loaded from disk or perhaps just *eval()*ed right there. This define-on-the-fly strategy is why it's called autoloading.

But that's only one possible approach. Another one is to just have the autoloaded method itself directly provide the requested service. When used in this way, you may think of autoloaded methods as "proxy" methods.

When Perl tries to call an undefined function in a particular package and that function is not defined, it looks for a function in that same package called AUTOLOAD. If one exists, it's called with the same arguments as the original function would have had. The fully-qualified name of the function is stored in that package's global variable `$AUTOLOAD`. Once called, the function can do anything it would like, including defining a new function by the right name, and then doing a really fancy kind of `goto` right to it, erasing itself from the call stack.

What does this have to do with objects? After all, we keep talking about functions, not methods. Well, since a method is just a function with an extra argument and some fancier semantics about where it's found, we can use autoloading for methods, too. Perl doesn't start looking for an AUTOLOAD method until it has exhausted the recursive hunt up through `@ISA`, though. Some programmers have even been known to define a UNIVERSAL::AUTOLOAD method to trap unresolved method calls to any kind of object.

**Autoloaded Data Methods**

You probably began to get a little suspicious about the duplicated code way back earlier when we first showed you the Person class, and then later the Employee class. Each method used to access the hash fields looked virtually identical. This should have tickled that great programming virtue, Impatience, but for the time, we let Laziness win out, and so did nothing. Proxy methods can cure this.

Instead of writing a new function every time we want a new data field, we'll use the autoload mechanism to generate (actually, mimic) methods on the fly. To verify that we're accessing a valid member, we will check against an `_permitted` (pronounced "under-permitted") field, which is a reference to a file-scoped lexical (like a C file static) hash of permitted fields in this record called `%fields`. Why the underscore? For the same reason as the _CENSUS field we once used: as a marker that means "for internal use only".

Here's what the module initialization code and class constructor will look like when taking this approach:

```
package Person;
use Carp;
use vars qw($AUTOLOAD);  # it's a package global

my %fields = (
    name        => undef,
    age         => undef,
    peers       => undef,
);

sub new {
    my $that  = shift;
    my $class = ref($that) || $that;
    my $self  = {
        _permitted => \%fields,
        %fields,
    };
    bless $self, $class;
    return $self;
}
```

If we wanted our record to have default values, we could fill those in where current we have `undef` in the

`%fields` hash.

Notice how we saved a reference to our class data on the object itself? Remember that it's important to access class data through the object itself instead of having any method reference `%fields` directly, or else you won't have a decent inheritance.

The real magic, though, is going to reside in our proxy method, which will handle all calls to undefined methods for objects of class Person (or subclasses of Person). It has to be called AUTOLOAD. Again, it's all caps because it's called for us implicitly by Perl itself, not by a user directly.

```
sub AUTOLOAD {
    my $self = shift;
    my $type = ref($self)
                or croak "$self is not an object";

    my $name = $AUTOLOAD;
    $name =~ s/.*://;   # strip fully-qualified portion

    unless (exists $self->{_permitted}->{$name} ) {
        croak "Can't access `$name' field in class $type";
    }

    if (@_) {
        return $self->{$name} = shift;
    } else {
        return $self->{$name};
    }
}
```

Pretty nifty, eh? All we have to do to add new data fields is modify `%fields`. No new functions need be written.

I could have avoided the `_permitted` field entirely, but I wanted to demonstrate how to store a reference to class data on the object so you wouldn't have to access that class data directly from an object method.

### Inherited Autoloaded Data Methods

But what about inheritance? Can we define our Employee class similarly? Yes, so long as we're careful enough.

Here's how to be careful:

```
package Employee;
use Person;
use strict;
use vars qw(@ISA);
@ISA = qw(Person);

my %fields = (
    id          => undef,
    salary      => undef,
);
```

```
sub new {
    my $that  = shift;
    my $class = ref($that) || $that;
    my $self = bless $that->SUPER::new(), $class;
    my($element);
    foreach $element (keys %fields) {
        $self->{_permitted}->{$element} = $fields{$element};
    }
    @{$self}{keys %fields} = values %fields;
    return $self;
}
```

Once we've done this, we don't even need to have an AUTOLOAD function in the Employee package, because we'll grab Person's version of that via inheritance, and it will all work out just fine.

**Metaclassical Tools**

Even though proxy methods can provide a more convenient approach to making more struct-like classes than tediously coding up data methods as functions, it still leaves a bit to be desired. For one thing, it means you have to handle bogus calls that you don't mean to trap via your proxy. It also means you have to be quite careful when dealing with inheritance, as detailed above.

Perl programmers have responded to this by creating several different class construction classes. These metaclasses are classes that create other classes. A couple worth looking at are Class::Struct and Alias. These and other related metaclasses can be found in the modules directory on CPAN.

### Class::Struct

One of the older ones is Class::Struct. In fact, its syntax and interface were sketched out long before perl5 even solidified into a real thing. What it does is provide you a way to ''declare'' a class as having objects whose fields are of a specific type. The function that does this is called, not surprisingly enough, *struct()*. Because structures or records are not base types in Perl, each time you want to create a class to provide a record-like data object, you yourself have to define a *new()* method, plus separate data-access methods for each of that record's fields. You'll quickly become bored with this process. The *Class::Struct::struct()* function alleviates this tedium.

Here's a simple example of using it:

```
use Class::Struct qw(struct);
use Jobbie;  # user-defined; see below

struct 'Fred' => {
    one        => '$',
    many       => '@',
    profession => Jobbie,  # calls Jobbie->new()
};

$ob = Fred->new;
$ob->one("hmmmm");

$ob->many(0, "here");
$ob->many(1, "you");
$ob->many(2, "go");
print "Just set: ", $ob->many(2), "\n";

$ob->profession->salary(10_000);
```

You can declare types in the struct to be basic Perl types, or user-defined types (classes). User types will be initialized by calling that class's *new()* method.

Here's a real-world example of using struct generation. Let's say you wanted to override Perl's idea of *gethostbyname()* and *gethostbyaddr()* so that they would return objects that acted like C structures. We don't care about high-falutin' OO gunk. All we want is for these objects to act like structs in the C sense.

```
use Socket;
use Net::hostent;
$h = gethostbyname("perl.com");  # object return
printf "perl.com's real name is %s, address %s\n",
    $h->name, inet_ntoa($h->addr);
```

Here's how to do this using the Class::Struct module. The crux is going to be this call:

```
struct 'Net::hostent' => [          # note bracket
    name       => '$',
    aliases    => '@',
    addrtype   => '$',
    'length'   => '$',
    addr_list  => '@',
 ];
```

Which creates object methods of those names and types. It even creates a *new()* method for us.

We could also have implemented our object this way:

```
struct 'Net::hostent' => {          # note brace
    name       => '$',
    aliases    => '@',
    addrtype   => '$',
    'length'   => '$',
    addr_list  => '@',
 };
```

and then Class::Struct would have used an anonymous hash as the object type, instead of an anonymous array. The array is faster and smaller, but the hash works out better if you eventually want to do inheritance. Since for this struct-like object we aren't planning on inheritance, this time we'll opt for better speed and size over better flexibility.

Here's the whole implementation:

```
package Net::hostent;
use strict;

BEGIN {
    use Exporter   ();
    use vars       qw(@EXPORT @EXPORT_OK %EXPORT_TAGS);
    @EXPORT      = qw(gethostbyname gethostbyaddr gethost);
    @EXPORT_OK   = qw(
                       $h_name         @h_aliases
                       $h_addrtype     $h_length
                       @h_addr_list    $h_addr
                   );
    %EXPORT_TAGS = ( FIELDS => [ @EXPORT_OK, @EXPORT ] );
}
use vars      @EXPORT_OK;

# Class::Struct forbids use of @ISA
sub import { goto &Exporter::import }
```

```
use Class::Struct qw(struct);
struct 'Net::hostent' => [
   name        => '$',
   aliases     => '@',
   addrtype    => '$',
   'length'    => '$',
   addr_list   => '@',
];

sub addr { shift->addr_list->[0] }

sub populate (@) {
    return unless @_;
    my $hob = new();  # Class::Struct made this!
    $h_name     =    $hob->[0]              = $_[0];
    @h_aliases  = @{ $hob->[1] } = split ' ', $_[1];
    $h_addrtype =    $hob->[2]              = $_[2];
    $h_length   =    $hob->[3]              = $_[3];
    $h_addr     =                             $_[4];
    @h_addr_list = @{ $hob->[4] } =         @_[ (4 .. $#_) ];
    return $hob;
}

sub gethostbyname ($)  { populate(CORE::gethostbyname(shift)) }

sub gethostbyaddr ($;$) {
    my ($addr, $addrtype);
    $addr = shift;
    require Socket unless @_;
    $addrtype = @_ ? shift : Socket::AF_INET();
    populate(CORE::gethostbyaddr($addr, $addrtype))
}

sub gethost($) {
    if ($_[0] =~ /^\d+(?:\.\d+(?:\.\d+(?:\.\d+)?)?)?$/) {
       require Socket;
       &gethostbyaddr(Socket::inet_aton(shift));
    } else {
       &gethostbyname;
    }
}

1;
```

We've snuck in quite a fair bit of other concepts besides just dynamic class creation, like overriding core functions, import/export bits, function prototyping, short-cut function call via `&whatever`, and function replacement with `goto &whatever`. These all mostly make sense from the perspective of a traditional module, but as you can see, we can also use them in an object module.

You can look at other object-based, struct-like overrides of core functions in the 5.004 release of Perl in File::stat, Net::hostent, Net::netent, Net::protoent, Net::servent, Time::gmtime, Time::localtime, User::grent, and User::pwent. These modules have a final component that's all lowercase, by convention reserved for compiler pragmas, because they affect the compilation and change a builtin function. They also have the type names that a C programmer would most expect.

### Data Members as Variables

If you're used to C++ objects, then you're accustomed to being able to get at an object's data members as simple variables from within a method. The Alias module provides for this, as well as a good bit more, such as the possibility of private methods that the object can call but folks outside the class cannot.

Here's an example of creating a Person using the Alias module. When you update these magical instance variables, you automatically update value fields in the hash. Convenient, eh?

```
package Person;

# this is the same as before...
sub new {
     my $that  = shift;
     my $class = ref($that) || $that;
     my $self = {
        NAME  => undef,
        AGE   => undef,
        PEERS => [],
    };
    bless($self, $class);
    return $self;
}

use Alias qw(attr);
use vars qw($NAME $AGE $PEERS);

sub name {
    my $self = attr shift;
    if (@_) { $NAME = shift; }
    return    $NAME;
}

sub age {
    my $self = attr shift;
    if (@_) { $AGE = shift; }
    return    $AGE;
}

sub peers {
    my $self = attr shift;
    if (@_) { @PEERS = @_; }
    return    @PEERS;
}

sub exclaim {
    my $self = attr shift;
    return sprintf "Hi, I'm %s, age %d, working with %s",
        $NAME, $AGE, join(", ", @PEERS);
}

sub happy_birthday {
    my $self = attr shift;
    return ++$AGE;
}
```

The need for the `use vars` declaration is because what Alias does is play with package globals with the same name as the fields. To use globals while `use strict` is in effect, you have to predeclare them. These package variables are localized to the block enclosing the *attr()* call just as if you'd used a *local()* on them. However, that means that they're still considered global variables with temporary values, just as with any other *local()*.

It would be nice to combine Alias with something like Class::Struct or Class::MethodMaker.

**NOTES**

### Object Terminology

In the various OO literature, it seems that a lot of different words are used to describe only a few different concepts. If you're not already an object programmer, then you don't need to worry about all these fancy words. But if you are, then you might like to know how to get at the same concepts in Perl.

For example, it's common to call an object an *instance* of a class and to call those objects' methods *instance methods*. Data fields peculiar to each object are often called *instance data* or *object attributes*, and data fields common to all members of that class are *class data*, *class attributes*, or *static data members*.

Also, *base class*, *generic class*, and *superclass* all describe the same notion, whereas *derived class*, *specific class*, and *subclass* describe the other related one.

C++ programmers have *static methods* and *virtual methods*, but Perl only has *class methods* and *object methods*. Actually, Perl only has methods. Whether a method gets used as a class or object method is by usage only. You could accidentally call a class method (one expecting a string argument) on an object (one expecting a reference), or vice versa.

From the C++ perspective, all methods in Perl are virtual. This, by the way, is why they are never checked for function prototypes in the argument list as regular builtin and user-defined functions can be.

Because a class is itself something of an object, Perl's classes can be taken as describing both a "class as meta-object" (also called *object factory*) philosophy and the "class as type definition" (*declaring* behaviour, not *defining* mechanism) idea. C++ supports the latter notion, but not the former.

## SEE ALSO

The following manpages will doubtless provide more background for this one: the *perlmod* manpage, the *perlref* manpage, the *perlobj* manpage, the *perlbot* manpage, the *perltie* manpage, and the *overload* manpage.

## AUTHOR AND COPYRIGHT



## COPYRIGHT

### Acknowledgments

Thanks to Larry Wall, Roderick Schertler, Gurusamy Sarathy, Dean Roehrich, Raphael Manfredi, Brent Halsey, Greg Bacon, Brad Appleton, and many others for their helpful comments.

**NAME**

perlobj – Perl objects

**DESCRIPTION**

First of all, you need to understand what references are in Perl. See the *perlref* manpage for that. Second, if you still find the following reference work too complicated, a tutorial on object-oriented programming in Perl can be found in the *perltoot* manpage.

If you're still with us, then here are three very simple definitions that you should find reassuring.

1. An object is simply a reference that happens to know which class it belongs to.
2. A class is simply a package that happens to provide methods to deal with object references.
3. A method is simply a subroutine that expects an object reference (or a package name, for class methods) as the first argument.

We'll cover these points now in more depth.

### An Object is Simply a Reference

Unlike say C++, Perl doesn't provide any special syntax for constructors. A constructor is merely a subroutine that returns a reference to something "blessed" into a class, generally the class that the subroutine is defined in. Here is a typical constructor:

```
package Critter;
sub new { bless {} }
```

That word `new` isn't special. You could have written a construct this way, too:

```
package Critter;
sub spawn { bless {} }
```

In fact, this might even be preferable, because the C++ programmers won't be tricked into thinking that `new` works in Perl as it does in C++. It doesn't. We recommend that you name your constructors whatever makes sense in the context of the problem you're solving. For example, constructors in the Tk extension to Perl are named after the widgets they create.

One thing that's different about Perl constructors compared with those in C++ is that in Perl, they have to allocate their own memory. (The other things is that they don't automatically call overridden base-class constructors.) The `{}` allocates an anonymous hash containing no key/value pairs, and returns it The *bless()* takes that reference and tells the object it references that it's now a Critter, and returns the reference. This is for convenience, because the referenced object itself knows that it has been blessed, and the reference to it could have been returned directly, like this:

```
sub new {
    my $self = {};
    bless $self;
    return $self;
}
```

In fact, you often see such a thing in more complicated constructors that wish to call methods in the class as part of the construction:

```
sub new {
    my $self = {};
    bless $self;
    $self->initialize();
    return $self;
}
```

If you care about inheritance (and you should; see the section on *Modules: Creation, Use, and Abuse* in the *perlmod* manpage), then you want to use the two-arg form of bless so that your constructors may be inherited:

```
sub new {
    my $class = shift;
    my $self = {};
    bless $self, $class;
    $self->initialize();
    return $self;
}
```

Or if you expect people to call not just `CLASS->new()` but also `$obj->new()`, then use something like this. The *initialize()* method used will be of whatever `$class` we blessed the object into:

```
sub new {
    my $this = shift;
    my $class = ref($this) || $this;
    my $self = {};
    bless $self, $class;
    $self->initialize();
    return $self;
}
```

Within the class package, the methods will typically deal with the reference as an ordinary reference. Outside the class package, the reference is generally treated as an opaque value that may be accessed only through the class's methods.

A constructor may re-bless a referenced object currently belonging to another class, but then the new class is responsible for all cleanup later. The previous blessing is forgotten, as an object may belong to only one class at a time. (Although of course it's free to inherit methods from many classes.) If you find yourself having to do this, the parent class is probably misbehaving, though.

A clarification: Perl objects are blessed. References are not. Objects know which package they belong to. References do not. The *bless()* function uses the reference to find the object. Consider the following example:

```
$a = {};
$b = $a;
bless $a, BLAH;
print "\$b is a ", ref($b), "\n";
```

This reports `$b` as being a BLAH, so obviously *bless()* operated on the object and not on the reference.

### A Class is Simply a Package

Unlike say C++, Perl doesn't provide any special syntax for class definitions. You use a package as a class by putting method definitions into the class.

There is a special array within each package called `@ISA`, which says where else to look for a method if you can't find it in the current package. This is how Perl implements inheritance. Each element of the `@ISA` array is just the name of another package that happens to be a class package. The classes are searched (depth first) for missing methods in the order that they occur in `@ISA`. The classes accessible through `@ISA` are known as base classes of the current class.

All classes implicitly inherit from class `UNIVERSAL` as their last base class. Several commonly used methods are automatically supplied in the UNIVERSAL class; see the section on *Default UNIVERSAL methods* for more details.

If a missing method is found in one of the base classes, it is cached in the current class for efficiency. Changing `@ISA` or defining new subroutines invalidates the cache and causes Perl to do the lookup again.

If neither the current class, its named base classes, nor the UNIVERSAL class contains the requested method, these three places are searched all over again, this time looking for a method named *AUTOLOAD()*. If an AUTOLOAD is found, this method is called on behalf of the missing method, setting the package global `$AUTOLOAD` to be the fully qualified name of the method that was intended to be called.

If none of that works, Perl finally gives up and complains.

Perl classes do method inheritance only. Data inheritance is left up to the class itself. By and large, this is not a problem in Perl, because most classes model the attributes of their object using an anonymous hash, which serves as its own little namespace to be carved up by the various classes that might want to do something with the object. The only problem with this is that you can't sure that you aren't using a piece of the hash that isn't already used. A reasonable workaround is to prepend your fieldname in the hash with the package name.

```
sub bump {
    my $self = shift;
    $self->{ __PACKAGE__ . ".count"}++;
}
```

### A Method is Simply a Subroutine

Unlike say C++, Perl doesn't provide any special syntax for method definition. (It does provide a little syntax for method invocation though. More on that later.) A method expects its first argument to be the object (reference) or package (string) it is being invoked on. There are just two types of methods, which we'll call class and instance. (Sometimes you'll hear these called static and virtual, in honor of the two C++ method types they most closely resemble.)

A class method expects a class name as the first argument. It provides functionality for the class as a whole, not for any individual object belonging to the class. Constructors are typically class methods. Many class methods simply ignore their first argument, because they already know what package they're in, and don't care what package they were invoked via. (These aren't necessarily the same, because class methods follow the inheritance tree just like ordinary instance methods.) Another typical use for class methods is to look up an object by name:

```
sub find {
    my ($class, $name) = @_;
    $objtable{$name};
}
```

An instance method expects an object reference as its first argument. Typically it shifts the first argument into a "self" or "this" variable, and then uses that as an ordinary reference.

```
sub display {
    my $self = shift;
    my @keys = @_ ? @_ : sort keys %$self;
    foreach $key (@keys) {
        print "\t$key => $self->{$key}\n";
    }
}
```

### Method Invocation

There are two ways to invoke a method, one of which you're already familiar with, and the other of which will look familiar. Perl 4 already had an "indirect object" syntax that you use when you say

```
print STDERR "help!!!\n";
```

This same syntax can be used to call either class or instance methods. We'll use the two methods defined above, the class method to lookup an object reference and the instance method to print out its attributes.

```
$fred = find Critter "Fred";
display $fred 'Height', 'Weight';
```

These could be combined into one statement by using a BLOCK in the indirect object slot:

```
display {find Critter "Fred"} 'Height', 'Weight';
```

For C++ fans, there's also a syntax using −> notation that does exactly the same thing. The parentheses are required if there are any arguments.

```
$fred = Critter->find("Fred");
$fred->display('Height', 'Weight');
```

or in one statement,

```
Critter->find("Fred")->display('Height', 'Weight');
```

There are times when one syntax is more readable, and times when the other syntax is more readable. The indirect object syntax is less cluttered, but it has the same ambiguity as ordinary list operators. Indirect object method calls are parsed using the same rule as list operators: "If it looks like a function, it is a function". (Presuming for the moment that you think two words in a row can look like a function name. C++ programmers seem to think so with some regularity, especially when the first word is "new".) Thus, the parentheses of

```
new Critter ('Barney', 1.5, 70)
```

are assumed to surround ALL the arguments of the method call, regardless of what comes after. Saying

```
new Critter ('Bam' x 2), 1.4, 45
```

would be equivalent to

```
Critter->new('Bam' x 2), 1.4, 45
```

which is unlikely to do what you want.

There are times when you wish to specify which class's method to use. In this case, you can call your method as an ordinary subroutine call, being sure to pass the requisite first argument explicitly:

```
$fred =  MyCritter::find("Critter", "Fred");
MyCritter::display($fred, 'Height', 'Weight');
```

Note however, that this does not do any inheritance. If you wish merely to specify that Perl should *START* looking for a method in a particular package, use an ordinary method call, but qualify the method name with the package like this:

```
$fred = Critter->MyCritter::find("Fred");
$fred->MyCritter::display('Height', 'Weight');
```

If you're trying to control where the method search begins *and* you're executing in the class itself, then you may use the SUPER pseudo class, which says to start looking in your base class's `@ISA` list without having to name it explicitly:

```
$self->SUPER::display('Height', 'Weight');
```

Please note that the `SUPER::` construct is meaningful *only* within the class.

Sometimes you want to call a method when you don't know the method name ahead of time. You can use the arrow form, replacing the method name with a simple scalar variable containing the method name:

```
$method = $fast ? "findfirst" : "findbest";
$fred->$method(@args);
```

### Default UNIVERSAL methods

The `UNIVERSAL` package automatically contains the following methods that are inherited by all other classes:

isa(CLASS)
: `isa` returns *true* if its object is blessed into a subclass of `CLASS`

  `isa` is also exportable and can be called as a sub with two arguments. This allows the ability to check what a reference points to. Example

```
use UNIVERSAL qw(isa);

if(isa($ref, 'ARRAY')) {
    #...
}
```

can(METHOD)
: `can` checks to see if its object has a method called `METHOD`, if it does then a reference to the sub is returned, if it does not then *undef* is returned.

VERSION( [NEED] )
: `VERSION` returns the version number of the class (package). If the NEED argument is given then it will check that the current version (as defined by the `$VERSION` variable in the given package) not less than NEED; it will die if this is not the case. This method is normally called as a class method. This method is called automatically by the `VERSION` form of `use`.

```
use A 1.2 qw(some imported subs);
# implies:
A->VERSION(1.2);
```

**NOTE:** `can` directly uses Perl's internal code for method lookup, and `isa` uses a very similar method and cache-ing strategy. This may cause strange effects if the Perl code dynamically changes `@ISA` in any package.

You may add other methods to the UNIVERSAL class via Perl or XS code. You do not need to `use UNIVERSAL` in order to make these methods available to your program. This is necessary only if you wish to have `isa` available as a plain subroutine in the current package.

### Destructors

When the last reference to an object goes away, the object is automatically destroyed. (This may even be after you exit, if you've stored references in global variables.) If you want to capture control just before the object is freed, you may define a DESTROY method in your class. It will automatically be called at the appropriate moment, and you can do any extra cleanup you need to do. Perl passes a reference to the object under destruction as the first (and only) argument. Beware that the reference is a read-only value, and cannot be modified by manipulating `$_[0]` within the destructor. The object itself (i.e. the thingy the reference points to, namely `${$_[0]}`, `@{$_[0]}`, `%{$_[0]}` etc.) is not similarly constrained.

If you arrange to re-bless the reference before the destructor returns, perl will again call the DESTROY method for the re-blessed object after the current one returns. This can be used for clean delegation of object destruction, or for ensuring that destructors in the base classes of your choosing get called. Explicitly calling DESTROY is also possible, but is usually never needed.

Do not confuse the foregoing with how objects *CONTAINED* in the current one are destroyed. Such objects will be freed and destroyed automatically when the current object is freed, provided no other references to them exist elsewhere.

### WARNING

While indirect object syntax may well be appealing to English speakers and to C++ programmers, be not seduced! It suffers from two grave problems.

The first problem is that an indirect object is limited to a name, a scalar variable, or a block, because it would have to do too much lookahead otherwise, just like any other postfix dereference in the language. (These are the same quirky rules as are used for the filehandle slot in functions like `print` and `printf`.) This can lead to horribly confusing precedence problems, as in these next two lines:

```
move $obj->{FIELD};                 # probably wrong!
move $ary[$i];                      # probably wrong!
```

Those actually parse as the very surprising:

```
$obj->move->{FIELD};                # Well, lookee here
$ary->move->[$i];                   # Didn't expect this one, eh?
```

Rather than what you might have expected:

```
$obj->{FIELD}->move();              # You should be so lucky.
$ary[$i]->move;                     # Yeah, sure.
```

The left side of "–>" is not so limited, because it's an infix operator, not a postfix operator.

As if that weren't bad enough, think about this: Perl must guess *at compile time* whether `name` and `move` above are functions or methods. Usually Perl gets it right, but when it doesn't it, you get a function call compiled as a method, or vice versa. This can introduce subtle bugs that are hard to unravel. For example, calling a method `new` in indirect notation— as C++ programmers are so wont to do— can be miscompiled into a subroutine call if there's already a `new` function in scope. You'd end up calling the current package's `new` as a subroutine, rather than the desired class's method. The compiler tries to cheat by remembering bareword `requires`, but the grief if it messes up just isn't worth the years of debugging it would likely take you to to track such subtle bugs down.

The infix arrow notation using "`->`" doesn't suffer from either of these disturbing ambiguities, so we recommend you use it exclusively.

### Summary

That's about all there is to it. Now you need just to go off and buy a book about object-oriented design methodology, and bang your forehead with it for the next six months or so.

### Two-Phased Garbage Collection

For most purposes, Perl uses a fast and simple reference-based garbage collection system. For this reason, there's an extra dereference going on at some level, so if you haven't built your Perl executable using your C compiler's `-O` flag, performance will suffer. If you *have* built Perl with `cc -O`, then this probably won't matter.

A more serious concern is that unreachable memory with a non-zero reference count will not normally get freed. Therefore, this is a bad idea:

```
{
    my $a;
    $a = \$a;
}
```

Even thought `$a` *should* go away, it can't. When building recursive data structures, you'll have to break the self-reference yourself explicitly if you don't care to leak. For example, here's a self-referential node such as one might use in a sophisticated tree structure:

```
sub new_node {
    my $self = shift;
    my $class = ref($self) || $self;
    my $node = {};
    $node->{LEFT} = $node->{RIGHT} = $node;
    $node->{DATA} = [ @_ ];
    return bless $node => $class;
}
```

If you create nodes like that, they (currently) won't go away unless you break their self reference yourself. (In other words, this is not to be construed as a feature, and you shouldn't depend on it.)

Almost.

When an interpreter thread finally shuts down (usually when your program exits), then a rather costly but complete mark-and-sweep style of garbage collection is performed, and everything allocated by that thread gets destroyed. This is essential to support Perl as an embedded or a multithreadable language. For example, this program demonstrates Perl's two-phased garbage collection:

```
#!/usr/bin/perl
package Subtle;

sub new {
    my $test;
    $test = \$test;
    warn "CREATING " . \$test;
    return bless \$test;
}

sub DESTROY {
    my $self = shift;
    warn "DESTROYING $self";
}

package main;

warn "starting program";
{
    my $a = Subtle->new;
    my $b = Subtle->new;
    $$a = 0;  # break selfref
    warn "leaving block";
}

warn "just exited block";
warn "time to die...";
exit;
```

When run as */tmp/test*, the following output is produced:

```
starting program at /tmp/test line 18.
CREATING SCALAR(0x8e5b8) at /tmp/test line 7.
CREATING SCALAR(0x8e57c) at /tmp/test line 7.
leaving block at /tmp/test line 23.
DESTROYING Subtle=SCALAR(0x8e5b8) at /tmp/test line 13.
just exited block at /tmp/test line 26.
time to die... at /tmp/test line 27.
DESTROYING Subtle=SCALAR(0x8e57c) during global destruction.
```

Notice that "global destruction" bit there? That's the thread garbage collector reaching the unreachable.

Objects are always destructed, even when regular refs aren't and in fact are destructed in a separate pass before ordinary refs just to try to prevent object destructors from using refs that have been themselves destructed. Plain refs are only garbage-collected if the destruct level is greater than 0. You can test the higher levels of global destruction by setting the PERL_DESTRUCT_LEVEL environment variable, presuming `-DDEBUGGING` was enabled during perl build time.

A more complete garbage collection strategy will be implemented at a future date.

In the meantime, the best solution is to create a non-recursive container class that holds a pointer to the self-referential data structure. Define a DESTROY method for the containing object's class that manually breaks the circularities in the self-referential structure.

## SEE ALSO

A kinder, gentler tutorial on object-oriented programming in Perl can be found in the *perltoot* manpage. You should also check out the *perlbot* manpage for other object tricks, traps, and tips, as well as the *perlmodlib* manpage for some style guides on constructing both modules and classes.

## NAME

perltie − how to hide an object class in a simple variable

## SYNOPSIS

```
tie VARIABLE, CLASSNAME, LIST

$object = tied VARIABLE

untie VARIABLE
```

## DESCRIPTION

Prior to release 5.0 of Perl, a programmer could use *dbmopen()* to connect an on-disk database in the standard Unix *dbm*(3x) format magically to a `%HASH` in their program. However, their Perl was either built with one particular dbm library or another, but not both, and you couldn't extend this mechanism to other packages or types of variables.

Now you can.

The *tie()* function binds a variable to a class (package) that will provide the implementation for access methods for that variable. Once this magic has been performed, accessing a tied variable automatically triggers method calls in the proper class. The complexity of the class is hidden behind magic methods calls. The method names are in ALL CAPS, which is a convention that Perl uses to indicate that they're called implicitly rather than explicitly—just like the *BEGIN()* and *END()* functions.

In the *tie()* call, `VARIABLE` is the name of the variable to be enchanted. `CLASSNAME` is the name of a class implementing objects of the correct type. Any additional arguments in the `LIST` are passed to the appropriate constructor method for that class—meaning *TIESCALAR()*, *TIEARRAY()*, *TIEHASH()*, or *TIEHANDLE()*. (Typically these are arguments such as might be passed to the *dbminit()* function of C.) The object returned by the ''new'' method is also returned by the *tie()* function, which would be useful if you wanted to access other methods in `CLASSNAME`. (You don't actually have to return a reference to a right ''type'' (e.g., HASH or `CLASSNAME`) so long as it's a properly blessed object.) You can also retrieve a reference to the underlying object using the *tied()* function.

Unlike *dbmopen()*, the *tie()* function will not `use` or `require` a module for you—you need to do that explicitly yourself.

### Tying Scalars

A class implementing a tied scalar should define the following methods: TIESCALAR, FETCH, STORE, and possibly DESTROY.

Let's look at each in turn, using as an example a tie class for scalars that allows the user to do something like:

```
tie $his_speed, 'Nice', getppid();
tie $my_speed,  'Nice', $$;
```

And now whenever either of those variables is accessed, its current system priority is retrieved and returned. If those variables are set, then the process's priority is changed!

We'll use Jarkko Hietaniemi <*jhi@iki.fi*>'s BSD::Resource class (not included) to access the PRIO_PROCESS, PRIO_MIN, and PRIO_MAX constants from your system, as well as the *getpriority()* and *setpriority()* system calls. Here's the preamble of the class.

```
package Nice;
use Carp;
use BSD::Resource;
use strict;
$Nice::DEBUG = 0 unless defined $Nice::DEBUG;
```

TIESCALAR classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference to a new scalar (probably anonymous) that it's creating. For example:

```
sub TIESCALAR {
    my $class = shift;
    my $pid = shift || $$; # 0 means me

    if ($pid !~ /^\d+$/) {
        carp "Nice::Tie::Scalar got non-numeric pid $pid" if $^W;
        return undef;
    }

    unless (kill 0, $pid) { # EPERM or ERSCH, no doubt
        carp "Nice::Tie::Scalar got bad pid $pid: $!" if $^W;
        return undef;
    }

    return bless \$pid, $class;
}
```

This tie class has chosen to return an error rather than raising an exception if its constructor should fail. While this is how *dbmopen()* works, other classes may well not wish to be so forgiving. It checks the global variable `$^W` to see whether to emit a bit of noise anyway.

FETCH this

This method will be triggered every time the tied variable is accessed (read). It takes no arguments beyond its self reference, which is the object representing the scalar we're dealing with. Because in this case we're using just a SCALAR ref for the tied scalar object, a simple $$self allows the method to get at the real value stored there. In our example below, that real value is the process ID to which we've tied our variable.

```
sub FETCH {
    my $self = shift;
    confess "wrong type" unless ref $self;
    croak "usage error" if @_;
    my $nicety;
    local($!) = 0;
    $nicety = getpriority(PRIO_PROCESS, $$self);
    if ($!) { croak "getpriority failed: $!" }
    return $nicety;
}
```

This time we've decided to blow up (raise an exception) if the renice fails— there's no place for us to return an error otherwise, and it's probably the right thing to do.

STORE this, value

This method will be triggered every time the tied variable is set (assigned). Beyond its self reference, it also expects one (and only one) argument— the new value the user is trying to assign.

```
sub STORE {
    my $self = shift;
    confess "wrong type" unless ref $self;
    my $new_nicety = shift;
    croak "usage error" if @_;
```

```
        if ($new_nicety < PRIO_MIN) {
            carp sprintf
              "WARNING: priority %d less than minimum system priority %d",
                  $new_nicety, PRIO_MIN if $^W;
            $new_nicety = PRIO_MIN;
        }

        if ($new_nicety > PRIO_MAX) {
            carp sprintf
              "WARNING: priority %d greater than maximum system priority %d",
                  $new_nicety, PRIO_MAX if $^W;
            $new_nicety = PRIO_MAX;
        }

        unless (defined setpriority(PRIO_PROCESS, $$self, $new_nicety)) {
            confess "setpriority failed: $!";
        }
        return $new_nicety;
    }
```

DESTROY this

This method will be triggered when the tied variable needs to be destructed. As with other object classes, such a method is seldom necessary, because Perl deallocates its moribund object's memory for you automatically—this isn't C++, you know. We'll use a DESTROY method here for debugging purposes only.

```
sub DESTROY {
    my $self = shift;
    confess "wrong type" unless ref $self;
    carp "[ Nice::DESTROY pid $$self ]" if $Nice::DEBUG;
}
```

That's about all there is to it. Actually, it's more than all there is to it, because we've done a few nice things here for the sake of completeness, robustness, and general aesthetics. Simpler TIESCALAR classes are certainly possible.

### Tying Arrays

A class implementing a tied ordinary array should define the following methods: TIEARRAY, FETCH, STORE, FETCHSIZE, STORESIZE and perhaps DESTROY.

FETCHSIZE and STORESIZE are used to provide `$#array` and equivalent `scalar(@array)` access.

The methods POP, PUSH, SHIFT, UNSHIFT, SPLICE are required if the perl operator with the corresponding (but lowercase) name is to operate on the tied array. The **Tie::Array** class can be used as a base class to implement these in terms of the basic five methods above.

In addition EXTEND will be called when perl would have pre-extended allocation in a real array.

This means that tied arrays are now *complete*. The example below needs upgrading to illustrate this. (The documentation in **Tie::Array** is more complete.)

For this discussion, we'll implement an array whose indices are fixed at its creation. If you try to access anything beyond those bounds, you'll take an exception. For example:

```
require Bounded_Array;
tie @ary, 'Bounded_Array', 2;
$| = 1;
for $i (0 .. 10) {
    print "setting index $i: ";
    $ary[$i] = 10 * $i;
    $ary[$i] = 10 * $i;
    print "value of elt $i now $ary[$i]\n";
}
```

The preamble code for the class is as follows:

```
package Bounded_Array;
use Carp;
use strict;
```

TIEARRAY classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference through which the new array (probably an anonymous ARRAY ref) will be accessed.

In our example, just to show you that you don't *really* have to return an ARRAY reference, we'll choose a HASH reference to represent our object. A HASH works out well as a generic record type: the `{BOUND}` field will store the maximum bound allowed, and the `{ARRAY}` field will hold the true ARRAY ref. If someone outside the class tries to dereference the object returned (doubtless thinking it an ARRAY ref), they'll blow up. This just goes to show you that you should respect an object's privacy.

```
sub TIEARRAY {
    my $class = shift;
    my $bound = shift;
    confess "usage: tie(\@ary, 'Bounded_Array', max_subscript)"
        if @_ || $bound =~ /\D/;
    return bless {
        BOUND => $bound,
        ARRAY => [],
    }, $class;
}
```

FETCH this, index

This method will be triggered every time an individual element the tied array is accessed (read). It takes one argument beyond its self reference: the index whose value we're trying to fetch.

```
sub FETCH {
  my($self,$idx) = @_;
  if ($idx > $self->{BOUND}) {
    confess "Array OOB: $idx > $self->{BOUND}";
  }
  return $self->{ARRAY}[$idx];
}
```

As you may have noticed, the name of the FETCH method (et al.) is the same for all accesses, even though the constructors differ in names (TIESCALAR vs TIEARRAY). While in theory you could have the same class servicing several tied types, in practice this becomes cumbersome, and it's easiest to keep them at simply one tie type per class.

STORE this, index, value

This method will be triggered every time an element in the tied array is set (written). It takes two arguments beyond its self reference: the index at which we're trying to store something and the value we're trying to put there. For example:

```
sub STORE {
  my($self, $idx, $value) = @_;
  print "[STORE $value at $idx]\n" if _debug;
  if ($idx > $self->{BOUND} ) {
    confess "Array OOB: $idx > $self->{BOUND}";
  }
  return $self->{ARRAY}[$idx] = $value;
}
```

DESTROY this

This method will be triggered when the tied variable needs to be destructed. As with the scalar tie class, this is almost never needed in a language that does its own garbage collection, so this time we'll just leave it out.

The code we presented at the top of the tied array class accesses many elements of the array, far more than we've set the bounds to. Therefore, it will blow up once they try to access beyond the 2nd element of `@ary`, as the following output demonstrates:

```
setting index 0: value of elt 0 now 0
setting index 1: value of elt 1 now 10
setting index 2: value of elt 2 now 20
setting index 3: Array OOB: 3 > 2 at Bounded_Array.pm line 39
        Bounded_Array::FETCH called at testba line 12
```

**Tying Hashes**

As the first Perl data type to be tied (see *dbmopen()*), hashes have the most complete and useful *tie()* implementation. A class implementing a tied hash should define the following methods: TIEHASH is the constructor. FETCH and STORE access the key and value pairs. EXISTS reports whether a key is present in the hash, and DELETE deletes one. CLEAR empties the hash by deleting all the key and value pairs. FIRSTKEY and NEXTKEY implement the *keys()* and *each()* functions to iterate over all the keys. And DESTROY is called when the tied variable is garbage collected.

If this seems like a lot, then feel free to inherit from merely the standard Tie::Hash module for most of your methods, redefining only the interesting ones. See the *Tie::Hash* manpage for details.

Remember that Perl distinguishes between a key not existing in the hash, and the key existing in the hash but having a corresponding value of `undef`. The two possibilities can be tested with the `exists()` and `defined()` functions.

Here's an example of a somewhat interesting tied hash class: it gives you a hash representing a particular user's dot files. You index into the hash with the name of the file (minus the dot) and you get back that dot file's contents. For example:

```
use DotFiles;
tie %dot, 'DotFiles';
if ( $dot{profile} =~ /MANPATH/ ||
     $dot{login}   =~ /MANPATH/ ||
     $dot{cshrc}   =~ /MANPATH/    )
{
    print "you seem to set your MANPATH\n";
}
```

Or here's another sample of using our tied class:

```
tie %him, 'DotFiles', 'daemon';
foreach $f ( keys %him ) {
    printf "daemon dot file %s is size %d\n",
        $f, length $him{$f};
}
```

In our tied hash DotFiles example, we use a regular hash for the object containing several important fields, of which only the `{LIST}` field will be what the user thinks of as the real hash.

USER
: whose dot files this object represents

HOME
: where those dot files live

CLOBBER
: whether we should try to change or remove those dot files

LIST
: the hash of dot file names and content mappings

Here's the start of *Dotfiles.pm*:

```
package DotFiles;
use Carp;
sub whowasi { (caller(1))[3] . '()' }
my $DEBUG = 0;
sub debug { $DEBUG = @_ ? shift : 1 }
```

For our example, we want to be able to emit debugging info to help in tracing during development. We keep also one convenience function around internally to help print out warnings; *whowasi()* returns the function name that calls it.

Here are the methods for the DotFiles tied hash.

TIEHASH classname, LIST
: This is the constructor for the class. That means it is expected to return a blessed reference through which the new object (probably but not necessarily an anonymous hash) will be accessed.

  Here's the constructor:

```
sub TIEHASH {
    my $self = shift;
    my $user = shift || $>;
    my $dotdir = shift || '';
    croak "usage: @{[&whowasi]} [USER [DOTDIR]]" if @_;
    $user = getpwuid($user) if $user =~ /^\d+$/;
    my $dir = (getpwnam($user))[7]
            || croak "@{[&whowasi]}: no user $user";
    $dir .= "/$dotdir" if $dotdir;

    my $node = {
        USER    => $user,
        HOME    => $dir,
        LIST    => {},
        CLOBBER => 0,
    };
```

```
        opendir(DIR, $dir)
                || croak "@{[&whowasi]}: can't opendir $dir: $!";
        foreach $dot ( grep /^\./ && -f "$dir/$_", readdir(DIR)) {
            $dot =~ s/^\.//;
            $node->{LIST}{$dot} = undef;
        }
        closedir DIR;
        return bless $node, $self;
    }
```

It's probably worth mentioning that if you're going to filetest the return values out of a readdir, you'd better prepend the directory in question. Otherwise, because we didn't *chdir()* there, it would have been testing the wrong file.

FETCH this, key

This method will be triggered every time an element in the tied hash is accessed (read). It takes one argument beyond its self reference: the key whose value we're trying to fetch.

Here's the fetch for our DotFiles example.

```
    sub FETCH {
        carp &whowasi if $DEBUG;
        my $self = shift;
        my $dot = shift;
        my $dir = $self->{HOME};
        my $file = "$dir/.$dot";

        unless (exists $self->{LIST}->{$dot} || -f $file) {
            carp "@{[&whowasi]}: no $dot file" if $DEBUG;
            return undef;
        }

        if (defined $self->{LIST}->{$dot}) {
            return $self->{LIST}->{$dot};
        } else {
            return $self->{LIST}->{$dot} = `cat $dir/.$dot`;
        }
    }
```

It was easy to write by having it call the Unix *cat*(1) command, but it would probably be more portable to open the file manually (and somewhat more efficient). Of course, because dot files are a Unixy concept, we're not that concerned.

STORE this, key, value

This method will be triggered every time an element in the tied hash is set (written). It takes two arguments beyond its self reference: the index at which we're trying to store something, and the value we're trying to put there.

Here in our DotFiles example, we'll be careful not to let them try to overwrite the file unless they've called the *clobber()* method on the original object reference returned by *tie()*.

```
sub STORE {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $value = shift;
    my $file = $self->{HOME} . "/.$dot";
    my $user = $self->{USER};

    croak "@{[&whowasi]}: $file not clobberable"
        unless $self->{CLOBBER};

    open(F, "> $file") || croak "can't open $file: $!";
    print F $value;
    close(F);
}
```

If they wanted to clobber something, they might say:

```
$ob = tie %daemon_dots, 'daemon';
$ob->clobber(1);
$daemon_dots{signature} = "A true daemon\n";
```

Another way to lay hands on a reference to the underlying object is to use the *tied()* function, so they might alternately have set clobber using:

```
tie %daemon_dots, 'daemon';
tied(%daemon_dots)->clobber(1);
```

The clobber method is simply:

```
sub clobber {
    my $self = shift;
    $self->{CLOBBER} = @_ ? shift : 1;
}
```

DELETE this, key

This method is triggered when we remove an element from the hash, typically by using the *delete()* function. Again, we'll be careful to check whether they really want to clobber files.

```
sub DELETE   {
    carp &whowasi if $DEBUG;

    my $self = shift;
    my $dot = shift;
    my $file = $self->{HOME} . "/.$dot";
    croak "@{[&whowasi]}: won't remove file $file"
        unless $self->{CLOBBER};
    delete $self->{LIST}->{$dot};
    my $success = unlink($file);
    carp "@{[&whowasi]}: can't unlink $file: $!" unless $success;
    $success;
}
```

The value returned by DELETE becomes the return value of the call to *delete()*. If you want to emulate the normal behavior of *delete()*, you should return whatever FETCH would have returned for this key. In this example, we have chosen instead to return a value which tells the caller whether the file was successfully deleted.

CLEAR this

This method is triggered when the whole hash is to be cleared, usually by assigning the empty list to it.

In our example, that would remove all the user's dot files! It's such a dangerous thing that they'll have to set CLOBBER to something higher than 1 to make it happen.

```
sub CLEAR    {
    carp &whowasi if $DEBUG;
    my $self = shift;
    croak "@{[&whowasi]}: won't remove all dot files for $self->{USER}"
        unless $self->{CLOBBER} > 1;
    my $dot;
    foreach $dot ( keys %{$self->{LIST}}) {
        $self->DELETE($dot);
    }
}
```

EXISTS this, key

This method is triggered when the user uses the *exists()* function on a particular hash. In our example, we'll look at the `{LIST}` hash element for this:

```
sub EXISTS   {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    return exists $self->{LIST}->{$dot};
}
```

FIRSTKEY this

This method will be triggered when the user is going to iterate through the hash, such as via a *keys()* or *each()* call.

```
sub FIRSTKEY {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $a = keys %{$self->{LIST}};          # reset each() iterator
    each %{$self->{LIST}}
}
```

NEXTKEY this, lastkey

This method gets triggered during a *keys()* or *each()* iteration. It has a second argument which is the last key that had been accessed. This is useful if you're carrying about ordering or calling the iterator from more than one sequence, or not really storing things in a hash anywhere.

For our example, we're using a real hash so we'll do just the simple thing, but we'll have to go through the LIST field indirectly.

```
sub NEXTKEY  {
    carp &whowasi if $DEBUG;
    my $self = shift;
    return each %{ $self->{LIST} }
}
```

DESTROY this

This method is triggered when a tied hash is about to go out of scope. You don't really need it unless you're trying to add debugging or have auxiliary state to clean up. Here's a very simple function:

```
sub DESTROY  {
    carp &whowasi if $DEBUG;
}
```

Note that functions such as *keys()* and *values()* may return huge lists when used on large objects, like DBM files. You may prefer to use the *each()* function to iterate over such. Example:

```
# print out history file offsets
use NDBM_File;
tie(%HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

**Tying FileHandles**

This is partially implemented now.

A class implementing a tied filehandle should define the following methods: TIEHANDLE, at least one of PRINT, PRINTF, WRITE, READLINE, GETC, READ, and possibly CLOSE and DESTROY.

It is especially useful when perl is embedded in some other program, where output to STDOUT and STDERR may have to be redirected in some special way. See nvi and the Apache module for examples.

In our example we're going to create a shouting handle.

```
package Shout;
```

TIEHANDLE classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference of some sort. The reference can be used to hold some internal information.

```
sub TIEHANDLE { print "<shout>\n"; my $i; bless \$i, shift }
```

WRITE this, LIST

This method will be called when the handle is written to via the `syswrite` function.

```
sub WRITE {
    $r = shift;
    my($buf,$len,$offset) = @_;
    print "WRITE called, \$buf=$buf, \$len=$len, \$offset=$offset";
}
```

PRINT this, LIST

This method will be triggered every time the tied handle is printed to with the `print()` function. Beyond its self reference it also expects the list that was passed to the print function.

```
sub PRINT { $r = shift; $$r++; print join($,,map(uc($_),@_)),$\ }
```

PRINTF this, LIST

This method will be triggered every time the tied handle is printed to with the `printf()` function. Beyond its self reference it also expects the format and list that was passed to the printf function.

```
sub PRINTF {
    shift;
    my $fmt = shift;
    print sprintf($fmt, @_)."\n";
}
```

READ this, LIST

This method will be called when the handle is read from via the `read` or `sysread` functions.

```
sub READ {
    $r = shift;
    my($buf,$len,$offset) = @_;
    print "READ called, \$buf=$buf, \$len=$len, \$offset=$offset";
}
```

READLINE this

This method will be called when the handle is read from via <HANDLE>. The method should return undef when there is no more data.

```
sub READLINE { $r = shift; "PRINT called $$r times\n"; }
```

GETC this

This method will be called when the `getc` function is called.

```
sub GETC { print "Don't GETC, Get Perl"; return "a"; }
```

CLOSE this

This method will be called when the handle is closed via the `close` function.

```
sub CLOSE { print "CLOSE called.\n" }
```

DESTROY this

As with the other types of ties, this method will be called when the tied handle is about to be destroyed. This is useful for debugging and possibly cleaning up.

```
sub DESTROY { print "</shout>\n" }
```

Here's how to use our little example:

```
tie(*FOO,'Shout');
print FOO "hello\n";
$a = 4; $b = 6;
print FOO $a, " plus ", $b, " equals ", $a + $b, "\n";
print <FOO>;
```

### The `untie` Gotcha

If you intend making use of the object returned from either *tie()* or *tied()*, and if the tie's target class defines a destructor, there is a subtle gotcha you *must* guard against.

As setup, consider this (admittedly rather contrived) example of a tie; all it does is use a file to keep a log of the values assigned to a scalar.

```
package Remember;
```

```
use strict;
use IO::File;

sub TIESCALAR {
    my $class = shift;
    my $filename = shift;
    my $handle = new IO::File "> $filename"
                     or die "Cannot open $filename: $!\n";

    print $handle "The Start\n";
    bless {FH => $handle, Value => 0}, $class;
}

sub FETCH {
    my $self = shift;
    return $self->{Value};
}

sub STORE {
    my $self = shift;
    my $value = shift;
    my $handle = $self->{FH};
    print $handle "$value\n";
    $self->{Value} = $value;
}

sub DESTROY {
    my $self = shift;
    my $handle = $self->{FH};
    print $handle "The End\n";
    close $handle;
}

1;
```

Here is an example that makes use of this tie:

```
use strict;
use Remember;

my $fred;
tie $fred, 'Remember', 'myfile.txt';
$fred = 1;
$fred = 4;
$fred = 5;
untie $fred;
system "cat myfile.txt";
```

This is the output when it is executed:

```
The Start
1
4
5
The End
```

So far so good. Those of you who have been paying attention will have spotted that the tied object hasn't been used so far. So lets add an extra method to the Remember class to allow comments to be included in the file -- say, something like this:

```
sub comment {
    my $self = shift;
    my $text = shift;
    my $handle = $self->{FH};
    print $handle $text, "\n";
}
```

And here is the previous example modified to use the `comment` method (which requires the tied object):

```
use strict;
use Remember;

my ($fred, $x);
$x = tie $fred, 'Remember', 'myfile.txt';
$fred = 1;
$fred = 4;
comment $x "changing...";
$fred = 5;
untie $fred;
system "cat myfile.txt";
```

When this code is executed there is no output. Here's why:

When a variable is tied, it is associated with the object which is the return value of the TIESCALAR, TIEARRAY, or TIEHASH function. This object normally has only one reference, namely, the implicit reference from the tied variable. When *untie()* is called, that reference is destroyed. Then, as in the first example above, the object's destructor (DESTROY) is called, which is normal for objects that have no more valid references; and thus the file is closed.

In the second example, however, we have stored another reference to the tied object in `$x`. That means that when *untie()* gets called there will still be a valid reference to the object in existence, so the destructor is not called at that time, and thus the file is not closed. The reason there is no output is because the file buffers have not been flushed to disk.

Now that you know what the problem is, what can you do to avoid it? Well, the good old `-w` flag will spot any instances where you call *untie()* and there are still valid references to the tied object. If the second script above is run with the `-w` flag, Perl prints this warning message:

```
untie attempted while 1 inner references still exist
```

To get the script to work properly and silence the warning make sure there are no valid references to the tied object *before untie()* is called:

```
undef $x;
untie $fred;
```

**SEE ALSO**

See the *DB_File* manpage or the *Config* manpage for some interesting *tie()* implementations.

**BUGS**

Tied arrays are *incomplete*. They are also distinctly lacking something for the `$#ARRAY` access (which is hard, as it's an lvalue), as well as the other obvious array functions, like *push()*, *pop()*, *shift()*, *unshift()*, and *splice()*.

You cannot easily tie a multilevel data structure (such as a hash of hashes) to a dbm file. The first problem is that all but GDBM and Berkeley DB have size limitations, but beyond that, you also have problems with how references are to be represented on disk. One experimental module that does attempt to address this need partially is the MLDBM module. Check your nearest CPAN site as described in the *perlmodlib* manpage for source code to MLDBM.

**AUTHOR**

Tom Christiansen

TIEHANDLE by Sven Verdoolaege <*skimo@dns.ufsia.ac.be*> and Doug MacEachern <*dougm@osf.org*>

## NAME

perlbot – Bag'o Object Tricks (the BOT)

## DESCRIPTION

The following collection of tricks and hints is intended to whet curious appetites about such things as the use of instance variables and the mechanics of object and class relationships. The reader is encouraged to consult relevant textbooks for discussion of Object Oriented definitions and methodology. This is not intended as a tutorial for object-oriented programming or as a comprehensive guide to Perl's object oriented features, nor should it be construed as a style guide.

The Perl motto still holds: There's more than one way to do it.

## OO SCALING TIPS

1 Do not attempt to verify the type of `$self`. That'll break if the class is inherited, when the type of `$self` is valid but its package isn't what you expect. See rule 5.

2 If an object-oriented (OO) or indirect-object (IO) syntax was used, then the object is probably the correct type and there's no need to become paranoid about it. Perl isn't a paranoid language anyway. If people subvert the OO or IO syntax then they probably know what they're doing and you should let them do it. See rule 1.

3 Use the two-argument form of *bless()*. Let a subclass use your constructor. See the section on *INHERITING A CONSTRUCTOR*.

4 The subclass is allowed to know things about its immediate superclass, the superclass is allowed to know nothing about a subclass.

5 Don't be trigger happy with inheritance. A "using", "containing", or "delegation" relationship (some sort of aggregation, at least) is often more appropriate. See the section on *OBJECT RELATIONSHIPS*, the section on *USING RELATIONSHIP WITH SDBM*, and the section on *DELEGATION*.

6 The object is the namespace. Make package globals accessible via the object. This will remove the guess work about the symbol's home package. See the section on *CLASS CONTEXT AND THE OBJECT*.

7 IO syntax is certainly less noisy, but it is also prone to ambiguities that can cause difficult-to-find bugs. Allow people to use the sure-thing OO syntax, even if you don't like it.

8 Do not use function-call syntax on a method. You're going to be bitten someday. Someone might move that method into a superclass and your code will be broken. On top of that you're feeding the paranoia in rule 2.

9 Don't assume you know the home package of a method. You're making it difficult for someone to override that method. See the section on *THINKING OF CODE REUSE*.

## INSTANCE VARIABLES

An anonymous array or anonymous hash can be used to hold instance variables. Named parameters are also demonstrated.

```
package Foo;

sub new {
        my $type = shift;
        my %params = @_;
        my $self = {};
        $self->{'High'} = $params{'High'};
        $self->{'Low'}  = $params{'Low'};
        bless $self, $type;
}

package Bar;
```

```
sub new {
        my $type = shift;
        my %params = @_;
        my $self = [];
        $self->[0] = $params{'Left'};
        $self->[1] = $params{'Right'};
        bless $self, $type;
}

package main;

$a = Foo->new( 'High' => 42, 'Low' => 11 );
print "High=$a->{'High'}\n";
print "Low=$a->{'Low'}\n";

$b = Bar->new( 'Left' => 78, 'Right' => 40 );
print "Left=$b->[0]\n";
print "Right=$b->[1]\n";
```

### SCALAR INSTANCE VARIABLES

An anonymous scalar can be used when only one instance variable is needed.

```
package Foo;

sub new {
        my $type = shift;
        my $self;
        $self = shift;
        bless \$self, $type;
}

package main;

$a = Foo->new( 42 );
print "a=$$a\n";
```

### INSTANCE VARIABLE INHERITANCE

This example demonstrates how one might inherit instance variables from a superclass for inclusion in the new class. This requires calling the superclass's constructor and adding one's own instance variables to the new object.

```
package Bar;

sub new {
        my $type = shift;
        my $self = {};
        $self->{'buz'} = 42;
        bless $self, $type;
}

package Foo;
@ISA = qw( Bar );
```

```
sub new {
        my $type = shift;
        my $self = Bar->new;
        $self->{'biz'} = 11;
        bless $self, $type;
}

package main;

$a = Foo->new;
print "buz = ", $a->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

**OBJECT RELATIONSHIPS**

The following demonstrates how one might implement "containing" and "using" relationships between objects.

```
package Bar;

sub new {
        my $type = shift;
        my $self = {};
        $self->{'buz'} = 42;
        bless $self, $type;
}

package Foo;

sub new {
        my $type = shift;
        my $self = {};
        $self->{'Bar'} = Bar->new;
        $self->{'biz'} = 11;
        bless $self, $type;
}

package main;

$a = Foo->new;
print "buz = ", $a->{'Bar'}->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

**OVERRIDING SUPERCLASS METHODS**

The following example demonstrates how to override a superclass method and then call the overridden method. The **SUPER** pseudo-class allows the programmer to call an overridden superclass method without actually knowing where that method is defined.

```
package Buz;
sub goo { print "here's the goo\n" }

package Bar; @ISA = qw( Buz );
sub google { print "google here\n" }
```

```
package Baz;
sub mumble { print "mumbling\n" }

package Foo;
@ISA = qw( Bar Baz );

sub new {
        my $type = shift;
        bless [], $type;
}
sub grr { print "grumble\n" }
sub goo {
        my $self = shift;
        $self->SUPER::goo();
}
sub mumble {
        my $self = shift;
        $self->SUPER::mumble();
}
sub google {
        my $self = shift;
        $self->SUPER::google();
}

package main;

$foo = Foo->new;
$foo->mumble;
$foo->grr;
$foo->goo;
$foo->google;
```

**USING RELATIONSHIP WITH SDBM**

This example demonstrates an interface for the SDBM class. This creates a ‘‘using’’ relationship between the SDBM class and the new class Mydbm.

```
package Mydbm;

require SDBM_File;
require Tie::Hash;
@ISA = qw( Tie::Hash );
```

```
sub TIEHASH {
    my $type = shift;
    my $ref  = SDBM_File->new(@_);
    bless {'dbm' => $ref}, $type;
}
sub FETCH {
    my $self = shift;
    my $ref  = $self->{'dbm'};
    $ref->FETCH(@_);
}
sub STORE {
    my $self = shift;
    if (defined $_[0]){
        my $ref = $self->{'dbm'};
        $ref->STORE(@_);
    } else {
        die "Cannot STORE an undefined key in Mydbm\n";
    }
}

package main;
use Fcntl qw( O_RDWR O_CREAT );

tie %foo, "Mydbm", "Sdbm", O_RDWR|O_CREAT, 0640;
$foo{'bar'} = 123;
print "foo-bar = $foo{'bar'}\n";

tie %bar, "Mydbm", "Sdbm2", O_RDWR|O_CREAT, 0640;
$bar{'Cathy'} = 456;
print "bar-Cathy = $bar{'Cathy'}\n";
```

**THINKING OF CODE REUSE**

One strength of Object-Oriented languages is the ease with which old code can use new code. The following examples will demonstrate first how one can hinder code reuse and then how one can promote code reuse.

This first example illustrates a class which uses a fully-qualified method call to access the ''private'' method *BAZ()*. The second example will show that it is impossible to override the *BAZ()* method.

```
package FOO;

sub new {
        my $type = shift;
        bless {}, $type;
}
sub bar {
        my $self = shift;
        $self->FOO::private::BAZ;
}

package FOO::private;
```

```
    sub BAZ {
            print "in BAZ\n";
    }

    package main;

    $a = FOO->new;
    $a->bar;
```

Now we try to override the *BAZ()* method. We would like *FOO::bar()* to call *GOOP::BAZ()*, but this cannot happen because *FOO::bar()* explicitly calls *FOO::private::BAZ()*.

```
    package FOO;

    sub new {
            my $type = shift;
            bless {}, $type;
    }
    sub bar {
            my $self = shift;
            $self->FOO::private::BAZ;
    }

    package FOO::private;

    sub BAZ {
            print "in BAZ\n";
    }

    package GOOP;
    @ISA = qw( FOO );
    sub new {
            my $type = shift;
            bless {}, $type;
    }

    sub BAZ {
            print "in GOOP::BAZ\n";
    }

    package main;

    $a = GOOP->new;
    $a->bar;
```

To create reusable code we must modify class FOO, flattening class FOO::private. The next example shows a reusable class FOO which allows the method *GOOP::BAZ()* to be used in place of *FOO::BAZ()*.

```
    package FOO;
```

```
sub new {
        my $type = shift;
        bless {}, $type;
}
sub bar {
        my $self = shift;
        $self->BAZ;
}

sub BAZ {
        print "in BAZ\n";
}

package GOOP;
@ISA = qw( FOO );

sub new {
        my $type = shift;
        bless {}, $type;
}
sub BAZ {
        print "in GOOP::BAZ\n";
}

package main;

$a = GOOP->new;
$a->bar;
```

**CLASS CONTEXT AND THE OBJECT**

Use the object to solve package and class context problems. Everything a method needs should be available via the object or should be passed as a parameter to the method.

A class will sometimes have static or global data to be used by the methods. A subclass may want to override that data and replace it with new data. When this happens the superclass may not know how to find the new copy of the data.

This problem can be solved by using the object to define the context of the method. Let the method look in the object for a reference to the data. The alternative is to force the method to go hunting for the data ("Is it in my class, or in a subclass? Which subclass?"), and this can be inconvenient and will lead to hackery. It is better just to let the object tell the method where that data is located.

```
package Bar;

%fizzle = ( 'Password' => 'XYZZY' );

sub new {
        my $type = shift;
        my $self = {};
        $self->{'fizzle'} = \%fizzle;
        bless $self, $type;
}

sub enter {
        my $self = shift;
```

```
                # Don't try to guess if we should use %Bar::fizzle
                # or %Foo::fizzle.  The object already knows which
                # we should use, so just ask it.
                #
                my $fizzle = $self->{'fizzle'};

                print "The word is ", $fizzle->{'Password'}, "\n";
        }

        package Foo;
        @ISA = qw( Bar );

        %fizzle = ( 'Password' => 'Rumple' );

        sub new {
                my $type = shift;
                my $self = Bar->new;
                $self->{'fizzle'} = \%fizzle;
                bless $self, $type;
        }

        package main;

        $a = Bar->new;
        $b = Foo->new;
        $a->enter;
        $b->enter;
```

**INHERITING A CONSTRUCTOR**

An inheritable constructor should use the second form of *bless()* which allows blessing directly into a specified class. Notice in this example that the object will be a BAR not a FOO, even though the constructor is in class FOO.

```
        package FOO;

        sub new {
                my $type = shift;
                my $self = {};
                bless $self, $type;
        }

        sub baz {
                print "in FOO::baz()\n";
        }

        package BAR;
        @ISA = qw(FOO);

        sub baz {
                print "in BAR::baz()\n";
        }

        package main;
```

```
$a = BAR->new;
$a->baz;
```

**DELEGATION**

Some classes, such as SDBM_File, cannot be effectively subclassed because they create foreign objects. Such a class can be extended with some sort of aggregation technique such as the "using" relationship mentioned earlier or by delegation.

The following example demonstrates delegation using an *AUTOLOAD()* function to perform message-forwarding. This will allow the Mydbm object to behave exactly like an SDBM_File object. The Mydbm class could now extend the behavior by adding custom *FETCH()* and *STORE()* methods, if this is desired.

```
package Mydbm;

require SDBM_File;
require Tie::Hash;
@ISA = qw(Tie::Hash);

sub TIEHASH {
        my $type = shift;
        my $ref = SDBM_File->new(@_);
        bless {'delegate' => $ref};
}

sub AUTOLOAD {
        my $self = shift;

        # The Perl interpreter places the name of the
        # message in a variable called $AUTOLOAD.

        # DESTROY messages should never be propagated.
        return if $AUTOLOAD =~ /::DESTROY$/;

        # Remove the package name.
        $AUTOLOAD =~ s/^Mydbm:://;

        # Pass the message to the delegate.
        $self->{'delegate'}->$AUTOLOAD(@_);
}

package main;
use Fcntl qw( O_RDWR O_CREAT );

tie %foo, "Mydbm", "adbm", O_RDWR|O_CREAT, 0640;
$foo{'bar'} = 123;
print "foo-bar = $foo{'bar'}\n";
```

## NAME

perlipc − Perl interprocess communication (signals, fifos, pipes, safe subprocesses, sockets, and semaphores)

## DESCRIPTION

The basic IPC facilities of Perl are built out of the good old Unix signals, named pipes, pipe opens, the Berkeley socket routines, and SysV IPC calls. Each is used in slightly different situations.

### Signals

Perl uses a simple signal handling model: the `%SIG` hash contains names or references of user-installed signal handlers. These handlers will be called with an argument which is the name of the signal that triggered it. A signal may be generated intentionally from a particular keyboard sequence like control-C or control-Z, sent to you from another process, or triggered automatically by the kernel when special events transpire, like a child process exiting, your process running out of stack space, or hitting file size limit.

For example, to trap an interrupt signal, set up a handler like this. Do as little as you possibly can in your handler; notice how all we do is set a global variable and then raise an exception. That's because on most systems, libraries are not re-entrant; particularly, memory allocation and I/O routines are not. That means that doing nearly *anything* in your handler could in theory trigger a memory fault and subsequent core dump.

```
sub catch_zap {
    my $signame = shift;
    $shucks++;
    die "Somebody sent me a SIG$signame";
}
$SIG{INT} = 'catch_zap';  # could fail in modules
$SIG{INT} = \&catch_zap;  # best strategy
```

The names of the signals are the ones listed out by `kill -l` on your system, or you can retrieve them from the Config module. Set up an `@signame` list indexed by number to get the name and a `%signo` table indexed by name to get the number:

```
use Config;
defined $Config{sig_name} || die "No sigs?";
foreach $name (split(' ', $Config{sig_name})) {
    $signo{$name} = $i;
    $signame[$i] = $name;
    $i++;
}
```

So to check whether signal 17 and SIGALRM were the same, do just this:

```
print "signal #17 = $signame[17]\n";
if ($signo{ALRM}) {
    print "SIGALRM is $signo{ALRM}\n";
}
```

You may also choose to assign the strings `'IGNORE'` or `'DEFAULT'` as the handler, in which case Perl will try to discard the signal or do the default thing. Some signals can be neither trapped nor ignored, such as the KILL and STOP (but not the TSTP) signals. One strategy for temporarily ignoring signals is to use a *local()* statement, which will be automatically restored once your block is exited. (Remember that *local()* values are "inherited" by functions called from within that block.)

```
sub precious {
    local $SIG{INT} = 'IGNORE';
    &more_functions;
}
sub more_functions {
    # interrupts still ignored, for now...
}
```

Sending a signal to a negative process ID means that you send the signal to the entire Unix process-group. This code sends a hang-up signal to all processes in the current process group (and sets `$SIG{HUP}` to IGNORE so it doesn't kill itself):

```
{
    local $SIG{HUP} = 'IGNORE';
    kill HUP => -$$;
    # snazzy writing of: kill('HUP', -$$)
}
```

Another interesting signal to send is signal number zero. This doesn't actually affect another process, but instead checks whether it's alive or has changed its UID.

```
unless (kill 0 => $kid_pid) {
    warn "something wicked happened to $kid_pid";
}
```

You might also want to employ anonymous functions for simple signal handlers:

```
$SIG{INT} = sub { die "\nOutta here!\n" };
```

But that will be problematic for the more complicated handlers that need to reinstall themselves. Because Perl's signal mechanism is currently based on the *signal*(3) function from the C library, you may sometimes be so misfortunate as to run on systems where that function is "broken", that is, it behaves in the old unreliable SysV way rather than the newer, more reasonable BSD and POSIX fashion. So you'll see defensive people writing signal handlers like this:

```
sub REAPER {
    $waitedpid = wait;
    # loathe sysV: it makes us not only reinstate
    # the handler, but place it after the wait
    $SIG{CHLD} = \&REAPER;
}
$SIG{CHLD} = \&REAPER;
# now do something that forks...
```

or even the more elaborate:

```
use POSIX ":sys_wait_h";
sub REAPER {
    my $child;
    while ($child = waitpid(-1,WNOHANG)) {
        $Kid_Status{$child} = $?;
    }
    $SIG{CHLD} = \&REAPER;  # still loathe sysV
}
$SIG{CHLD} = \&REAPER;
# do something that forks...
```

Signal handling is also used for timeouts in Unix, While safely protected within an `eval{}` block, you set a signal handler to trap alarm signals and then schedule to have one delivered to you in some number of seconds. Then try your blocking operation, clearing the alarm when it's done but not before you've exited your `eval{}` block. If it goes off, you'll use *die()* to jump out of the block, much as you might using *longjmp()* or *throw()* in other languages.

Here's an example:

```
eval {
    local $SIG{ALRM} = sub { die "alarm clock restart" };
    alarm 10;
    flock(FH, 2);   # blocking write lock
    alarm 0;
};
if ($@ and $@ !~ /alarm clock restart/) { die }
```

For more complex signal handling, you might see the standard POSIX module. Lamentably, this is almost entirely undocumented, but the *t/lib/posix.t* file from the Perl source distribution has some examples in it.

**Named Pipes**

A named pipe (often referred to as a FIFO) is an old Unix IPC mechanism for processes communicating on the same machine. It works just like a regular, connected anonymous pipes, except that the processes rendezvous using a filename and don't have to be related.

To create a named pipe, use the Unix command *mknod* (1) or on some systems, *mkfifo* (1). These may not be in your normal path.

```
# system return val is backwards, so && not ||
#
$ENV{PATH} .= ":/etc:/usr/etc";
if  (      system('mknod',  $path, 'p')
        && system('mkfifo', $path) )
{
    die "mk{nod,fifo} $path failed";
}
```

A fifo is convenient when you want to connect a process to an unrelated one. When you open a fifo, the program will block until there's something on the other end.

For example, let's say you'd like to have your *.signature* file be a named pipe that has a Perl program on the other end. Now every time any program (like a mailer, news reader, finger program, etc.) tries to read from that file, the reading program will block and your program will supply the new signature. We'll use the pipe-checking file test **–p** to find out whether anyone (or anything) has accidentally removed our fifo.

```
chdir; # go home
$FIFO = '.signature';
$ENV{PATH} .= ":/etc:/usr/games";

while (1) {
    unless (-p $FIFO) {
        unlink $FIFO;
        system('mknod', $FIFO, 'p')
            && die "can't mknod $FIFO: $!";
    }
```

```
        # next line blocks until there's a reader
        open (FIFO, "> $FIFO") || die "can't write $FIFO: $!";
        print FIFO "John Smith (smith\@host.org)\n", `fortune -s`;
        close FIFO;
        sleep 2;    # to avoid dup signals
    }
```

**WARNING**

By installing Perl code to deal with signals, you're exposing yourself to danger from two things. First, few system library functions are re-entrant. If the signal interrupts while Perl is executing one function (like *malloc*(3) or *printf*(3)), and your signal handler then calls the same function again, you could get unpredictable behavior—often, a core dump. Second, Perl isn't itself re-entrant at the lowest levels. If the signal interrupts Perl while Perl is changing its own internal data structures, similarly unpredictable behaviour may result.

There are two things you can do, knowing this: be paranoid or be pragmatic. The paranoid approach is to do as little as possible in your signal handler. Set an existing integer variable that already has a value, and return. This doesn't help you if you're in a slow system call, which will just restart. That means you have to `die` to *longjump*(3) out of the handler. Even this is a little cavalier for the true paranoiac, who avoids `die` in a handler because the system *is* out to get you. The pragmatic approach is to say "I know the risks, but prefer the convenience", and to do anything you want in your signal handler, prepared to clean up core dumps now and again.

To forbid signal handlers altogether would bars you from many interesting programs, including virtually everything in this manpage, since you could no longer even write SIGCHLD handlers. Their dodginess is expected to be addresses in the 5.005 release.

## Using *open()* for IPC

Perl's basic *open()* statement can also be used for unidirectional interprocess communication by either appending or prepending a pipe symbol to the second argument to *open()*. Here's how to start something up in a child process you intend to write to:

```
    open(SPOOLER, "| cat -v | lpr -h 2>/dev/null")
                    || die "can't fork: $!";
    local $SIG{PIPE} = sub { die "spooler pipe broke" };
    print SPOOLER "stuff\n";
    close SPOOLER || die "bad spool: $! $?";
```

And here's how to start up a child process you intend to read from:

```
    open(STATUS, "netstat -an 2>&1 |")
                    || die "can't fork: $!";
    while (<STATUS>) {
        next if /^(tcp|udp)/;
        print;
    }
    close STATUS || die "bad netstat: $! $?";
```

If one can be sure that a particular program is a Perl script that is expecting filenames in `@ARGV`, the clever programmer can write something like this:

```
    % program f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

and irrespective of which shell it's called from, the Perl program will read from the file *f1*, the process *cmd1*, standard input (*tmpfile* in this case), the *f2* file, the *cmd2* command, and finally the *f3* file. Pretty nifty, eh?

You might notice that you could use backticks for much the same effect as opening a pipe for reading:

```
print grep { !/^(tcp|udp)/ } `netstat -an 2>&1`;
die "bad netstat" if $?;
```

While this is true on the surface, it's much more efficient to process the file one line or record at a time because then you don't have to read the whole thing into memory at once. It also gives you finer control of the whole process, letting you to kill off the child process early if you'd like.

Be careful to check both the *open()* and the *close()* return values. If you're *writing* to a pipe, you should also trap SIGPIPE. Otherwise, think of what happens when you start up a pipe to a command that doesn't exist: the *open()* will in all likelihood succeed (it only reflects the *fork()*'s success), but then your output will fail—spectacularly. Perl can't know whether the command worked because your command is actually running in a separate process whose *exec()* might have failed. Therefore, while readers of bogus commands return just a quick end of file, writers to bogus command will trigger a signal they'd better be prepared to handle. Consider:

```
open(FH, "|bogus")  or die "can't fork: $!";
print FH "bang\n"   or die "can't write: $!";
close FH            or die "can't close: $!";
```

That won't blow up until the close, and it will blow up with a SIGPIPE. To catch it, you could use this:

```
$SIG{PIPE} = 'IGNORE';
open(FH, "|bogus")  or die "can't fork: $!";
print FH "bang\n"   or die "can't write: $!";
close FH            or die "can't close: status=$?";
```

### Filehandles

Both the main process and any child processes it forks share the same STDIN, STDOUT, and STDERR filehandles. If both processes try to access them at once, strange things can happen. You'll certainly want to any stdio flush output buffers before forking. You may also want to close or reopen the filehandles for the child. You can get around this by opening your pipe with *open()*, but on some systems this means that the child process cannot outlive the parent.

### Background Processes

You can run a command in the background with:

```
system("cmd &");
```

The command's STDOUT and STDERR (and possibly STDIN, depending on your shell) will be the same as the parent's. You won't need to catch SIGCHLD because of the double-fork taking place (see below for more details).

### Complete Dissociation of Child from Parent

In some cases (starting server processes, for instance) you'll want to complete dissociate the child process from the parent. The easiest way is to use:

```
use POSIX qw(setsid);
setsid()            or die "Can't start a new session: $!";
```

However, you may not be on POSIX. The following process is reported to work on most Unixish systems. Non-Unix users should check their Your_OS::Process module for other solutions.

- Open /dev/tty and use the TIOCNOTTY ioctl on it. See the *tty(4)* manpage for details.
- Change directory to /
- Reopen STDIN, STDOUT, and STDERR so they're not connected to the old tty.
- Background yourself like this:

```
fork && exit;
```

- Ignore hangup signals in case you're running on a shell that doesn't automatically no-hup you:

```
$SIG{HUP} = 'IGNORE';       # or whatever you'd like
```

**Safe Pipe Opens**

Another interesting approach to IPC is making your single program go multiprocess and communicate between (or even amongst) yourselves. The *open()* function will accept a file argument of either `"-|"` or `"|-"` to do a very interesting thing: it forks a child connected to the filehandle you've opened. The child is running the same program as the parent. This is useful for safely opening a file when running under an assumed UID or GID, for example. If you open a pipe *to* minus, you can write to the filehandle you opened and your kid will find it in his STDIN. If you open a pipe *from* minus, you can read from the filehandle you opened whatever your kid writes to his STDOUT.

```
use English;
my $sleep_count = 0;

do {
    $pid = open(KID_TO_WRITE, "|-");
    unless (defined $pid) {
        warn "cannot fork: $!";
        die "bailing out" if $sleep_count++ > 6;
        sleep 10;
    }
} until defined $pid;

if ($pid) {  # parent
    print KID_TO_WRITE @some_data;
    close(KID_TO_WRITE) || warn "kid exited $?";
} else {     # child
    ($EUID, $EGID) = ($UID, $GID); # suid progs only
    open (FILE, "> /safe/file")
        || die "can't open /safe/file: $!";
    while (<STDIN>) {
        print FILE; # child's STDIN is parent's KID
    }
    exit;  # don't forget this
}
```

Another common use for this construct is when you need to execute something without the shell's interference. With *system()*, it's straightforward, but you can't use a pipe open or backticks safely. That's because there's no way to stop the shell from getting its hands on your arguments. Instead, use lower-level control to call *exec()* directly.

Here's a safe backtick or pipe open for read:

```
# add error processing as above
$pid = open(KID_TO_READ, "-|");

if ($pid) {   # parent
    while (<KID_TO_READ>) {
        # do something interesting
    }
    close(KID_TO_READ) || warn "kid exited $?";

} else {      # child
    ($EUID, $EGID) = ($UID, $GID); # suid only
    exec($program, @options, @args)
        || die "can't exec program: $!";
    # NOTREACHED
}
```

And here's a safe pipe open for writing:

```
# add error processing as above
$pid = open(KID_TO_WRITE, "|-");
$SIG{ALRM} = sub { die "whoops, $program pipe broke" };

if ($pid) {  # parent
    for (@data) {
        print KID_TO_WRITE;
    }
    close(KID_TO_WRITE) || warn "kid exited $?";

} else {     # child
    ($EUID, $EGID) = ($UID, $GID);
    exec($program, @options, @args)
        || die "can't exec program: $!";
    # NOTREACHED
}
```

Note that these operations are full Unix forks, which means they may not be correctly implemented on alien systems. Additionally, these are not true multithreading. If you'd like to learn more about threading, see the *modules* file mentioned below in the SEE ALSO section.

**Bidirectional Communication with Another Process**

While this works reasonably well for unidirectional communication, what about bidirectional communication? The obvious thing you'd like to do doesn't actually work:

```
open(PROG_FOR_READING_AND_WRITING, "| some program |")
```

and if you forget to use the **−w** flag, then you'll miss out entirely on the diagnostic message:

```
Can't do bidirectional pipe at -e line 1.
```

If you really want to, you can use the standard *open2()* library function to catch both ends. There's also an *open3()* for tridirectional I/O so you can also catch your child's STDERR, but doing so would then require an awkward *select()* loop and wouldn't allow you to use normal Perl input operations.

If you look at its source, you'll see that *open2()* uses low-level primitives like Unix *pipe()* and *exec()* calls to create all the connections. While it might have been slightly more efficient by using *socketpair()*, it would have then been even less portable than it already is. The *open2()* and *open3()* functions are unlikely to work anywhere except on a Unix system or some other one purporting to be POSIX compliant.

Here's an example of using *open2()*:

```
use FileHandle;
use IPC::Open2;
$pid = open2(*Reader, *Writer, "cat -u -n" );
Writer->autoflush(); # default here, actually
print Writer "stuff\n";
$got = <Reader>;
```

The problem with this is that Unix buffering is really going to ruin your day. Even though your `Writer` filehandle is auto-flushed, and the process on the other end will get your data in a timely manner, you can't usually do anything to force it to give it back to you in a similarly quick fashion. In this case, we could, because we gave *cat* a **–u** flag to make it unbuffered. But very few Unix commands are designed to operate over pipes, so this seldom works unless you yourself wrote the program on the other end of the double-ended pipe.

A solution to this is the nonstandard *Comm.pl* library. It uses pseudo-ttys to make your program behave more reasonably:

```
require 'Comm.pl';
$ph = open_proc('cat -n');
for (1..10) {
    print $ph "a line\n";
    print "got back ", scalar <$ph>;
}
```

This way you don't have to have control over the source code of the program you're using. The *Comm* library also has *expect()* and *interact()* functions. Find the library (and we hope its successor *IPC::Chat*) at your nearest CPAN archive as detailed in the SEE ALSO section below.

The newer Expect.pm module from CPAN also addresses this kind of thing. This module requires two other modules from CPAN: IO::Pty and IO::Stty. It sets up a pseudo-terminal to interact with programs that insist on using talking to the terminal device driver. If your system is amongst those supported, this may be your best bet.

### Bidirectional Communication with Yourself

If you want, you may make low-level *pipe()* and *fork()* to stitch this together by hand. This example only talks to itself, but you could reopen the appropriate handles to STDIN and STDOUT and call other processes.

```
#!/usr/bin/perl -w
# pipe1 - bidirectional communication using two pipe pairs
#         designed for the socketpair-challenged
use IO::Handle;     # thousands of lines just for autoflush :-(
pipe(PARENT_RDR, CHILD_WTR);                # XXX: failure?
pipe(CHILD_RDR,  PARENT_WTR);               # XXX: failure?
CHILD_WTR->autoflush(1);
PARENT_WTR->autoflush(1);
```

```
if ($pid = fork) {
    close PARENT_RDR; close PARENT_WTR;
    print CHILD_WTR "Parent Pid $$ is sending this\n";
    chomp($line = <CHILD_RDR>);
    print "Parent Pid $$ just read this: `$line'\n";
    close CHILD_RDR; close CHILD_WTR;
    waitpid($pid,0);
} else {
    die "cannot fork: $!" unless defined $pid;
    close CHILD_RDR; close CHILD_WTR;
    chomp($line = <PARENT_RDR>);
    print "Child Pid $$ just read this: `$line'\n";
    print PARENT_WTR "Child Pid $$ is sending this\n";
    close PARENT_RDR; close PARENT_WTR;
    exit;
}
```

But you don't actually have to make two pipe calls. If you have the *socketpair()* system call, it will do this all for you.

```
#!/usr/bin/perl -w
# pipe2 - bidirectional communication using socketpair
#   "the best ones always go both ways"

use Socket;
use IO::Handle;     # thousands of lines just for autoflush :-(
# We say AF_UNIX because although *_LOCAL is the
# POSIX 1003.1g form of the constant, many machines
# still don't have it.
socketpair(CHILD, PARENT, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
                            or  die "socketpair: $!";

CHILD->autoflush(1);
PARENT->autoflush(1);

if ($pid = fork) {
    close PARENT;
    print CHILD "Parent Pid $$ is sending this\n";
    chomp($line = <CHILD>);
    print "Parent Pid $$ just read this: `$line'\n";
    close CHILD;
    waitpid($pid,0);
} else {
    die "cannot fork: $!" unless defined $pid;
    close CHILD;
    chomp($line = <PARENT>);
    print "Child Pid $$ just read this: `$line'\n";
    print PARENT "Child Pid $$ is sending this\n";
    close PARENT;
    exit;
}
```

**Sockets: Client/Server Communication**

While not limited to Unix-derived operating systems (e.g., WinSock on PCs provides socket support, as do some VMS libraries), you may not have sockets on your system, in which case this section probably isn't going to do you much good. With sockets, you can do both virtual circuits (i.e., TCP streams) and datagrams (i.e., UDP packets). You may be able to do even more depending on your system.

The Perl function calls for dealing with sockets have the same names as the corresponding system calls in C, but their arguments tend to differ for two reasons: first, Perl filehandles work differently than C file descriptors. Second, Perl already knows the length of its strings, so you don't need to pass that information.

One of the major problems with old socket code in Perl was that it used hard-coded values for some of the constants, which severely hurt portability. If you ever see code that does anything like explicitly setting `$AF_INET = 2`, you know you're in for big trouble: An immeasurably superior approach is to use the `Socket` module, which more reliably grants access to various constants and functions you'll need.

If you're not writing a server/client for an existing protocol like NNTP or SMTP, you should give some thought to how your server will know when the client has finished talking, and vice-versa. Most protocols are based on one-line messages and responses (so one party knows the other has finished when a "\n" is received) or multi-line messages and responses that end with a period on an empty line ("\n.\n" terminates a message/response).

### Internet Line Terminators

The Internet line terminator is "\015\012". Under ASCII variants of Unix, that could usually be written as "\r\n", but under other systems, "\r\n" might at times be "\015\015\012", "\012\012\015", or something completely different. The standards specify writing "\015\012" to be conformant (be strict in what you provide), but they also recommend accepting a lone "\012" on input (but be lenient in what you require). We haven't always been very good about that in the code in this manpage, but unless you're on a Mac, you'll probably be ok.

### Internet TCP Clients and Servers

Use Internet-domain sockets when you want to do client-server communication that might extend to machines outside of your own system.

Here's a sample TCP client using Internet-domain sockets:

```
#!/usr/bin/perl -w
use strict;
use Socket;
my ($remote,$port, $iaddr, $paddr, $proto, $line);

$remote  = shift || 'localhost';
$port    = shift || 2345;  # random port
if ($port =~ /\D/) { $port = getservbyname($port, 'tcp') }
die "No port" unless $port;
$iaddr   = inet_aton($remote)               || die "no host: $remote";
$paddr   = sockaddr_in($port, $iaddr);

$proto   = getprotobyname('tcp');
socket(SOCK, PF_INET, SOCK_STREAM, $proto)  || die "socket: $!";
connect(SOCK, $paddr)    || die "connect: $!";
while (defined($line = <SOCK>)) {
    print $line;
}

close (SOCK)            || die "close: $!";
exit;
```

And here's a corresponding server to go along with it. We'll leave the address as INADDR_ANY so that the kernel can choose the appropriate interface on multihomed hosts. If you want sit on a particular interface (like the external side of a gateway or firewall machine), you should fill this in with your real address instead.

```
#!/usr/bin/perl -Tw
use strict;
BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
use Socket;
use Carp;
$EOL = "\015\012";

sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }

my $port = shift || 2345;
my $proto = getprotobyname('tcp');
$port = $1 if $port =~ /(\d+)/; # untaint port number

socket(Server, PF_INET, SOCK_STREAM, $proto)        || die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR,
                                    pack("l", 1))   || die "setsockopt: $!";
bind(Server, sockaddr_in($port, INADDR_ANY))        || die "bind: $!";
listen(Server,SOMAXCONN)                            || die "listen: $!";

logmsg "server started on port $port";

my $paddr;

$SIG{CHLD} = \&REAPER;

for ( ; $paddr = accept(Client,Server); close Client) {
    my($port,$iaddr) = sockaddr_in($paddr);
    my $name = gethostbyaddr($iaddr,AF_INET);

    logmsg "connection from $name [",
            inet_ntoa($iaddr), "]
            at port $port";

    print Client "Hello there, $name, it's now ",
                    scalar localtime, $EOL;
}
```

And here's a multithreaded version. It's multithreaded in that like most typical servers, it spawns (forks) a slave server to handle the client request so that the master server can quickly go back to service a new client.

```
#!/usr/bin/perl -Tw
use strict;
BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
use Socket;
use Carp;
$EOL = "\015\012";

sub spawn;  # forward declaration
sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }
```

```
my $port = shift || 2345;
my $proto = getprotobyname('tcp');
$port = $1 if $port =~ /(\d+)/; # untaint port number

socket(Server, PF_INET, SOCK_STREAM, $proto)        || die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR,
                                    pack("l", 1))   || die "setsockopt: $!";
bind(Server, sockaddr_in($port, INADDR_ANY))        || die "bind: $!";
listen(Server,SOMAXCONN)                            || die "listen: $!";

logmsg "server started on port $port";

my $waitedpid = 0;
my $paddr;

sub REAPER {
    $waitedpid = wait;
    $SIG{CHLD} = \&REAPER;  # loathe sysV
    logmsg "reaped $waitedpid" . ($? ? " with exit $?" : '');
}

$SIG{CHLD} = \&REAPER;

for ( $waitedpid = 0;
      ($paddr = accept(Client,Server)) || $waitedpid;
      $waitedpid = 0, close Client)
{
    next if $waitedpid and not $paddr;
    my($port,$iaddr) = sockaddr_in($paddr);
    my $name = gethostbyaddr($iaddr,AF_INET);

    logmsg "connection from $name [",
            inet_ntoa($iaddr), "]
            at port $port";

    spawn sub {
        print "Hello there, $name, it's now ", scalar localtime, $EOL;
        exec '/usr/games/fortune'           # XXX: `wrong' line terminators
            or confess "can't exec fortune: $!";
    };

}

sub spawn {
    my $coderef = shift;

    unless (@_ == 0 && $coderef && ref($coderef) eq 'CODE') {
        confess "usage: spawn CODEREF";
    }
```

```
        my $pid;
        if (!defined($pid = fork)) {
            logmsg "cannot fork: $!";
            return;
        } elsif ($pid) {
            logmsg "begat $pid";
            return; # I'm the parent
        }
        # else I'm the child -- go spawn

        open(STDIN,  "<&Client")   || die "can't dup client to stdin";
        open(STDOUT, ">&Client")   || die "can't dup client to stdout";
        ## open(STDERR, ">&STDOUT") || die "can't dup stdout to stderr";
        exit &$coderef();
    }
```

This server takes the trouble to clone off a child version via *fork()* for each incoming request. That way it can handle many requests at once, which you might not always want. Even if you don't *fork()*, the *listen()* will allow that many pending connections. Forking servers have to be particularly careful about cleaning up their dead children (called "zombies" in Unix parlance), because otherwise you'll quickly fill up your process table.

We suggest that you use the **−T** flag to use taint checking (see the *perlsec* manpage) even if we aren't running setuid or setgid. This is always a good idea for servers and other programs run on behalf of someone else (like CGI scripts), because it lessens the chances that people from the outside will be able to compromise your system.

Let's look at another TCP client. This one connects to the TCP "time" service on a number of different machines and shows how far their clocks differ from the system on which it's being run:

```
    #!/usr/bin/perl  -w
    use strict;
    use Socket;

    my $SECS_of_70_YEARS = 2208988800;
    sub ctime { scalar localtime(shift) }

    my $iaddr = gethostbyname('localhost');
    my $proto = getprotobyname('tcp');
    my $port = getservbyname('time', 'tcp');
    my $paddr = sockaddr_in(0, $iaddr);
    my($host);

    $| = 1;
    printf "%-24s %8s %s\n",  "localhost", 0, ctime(time());
```

```
foreach $host (@ARGV) {
    printf "%-24s ", $host;
    my $hisiaddr = inet_aton($host)     || die "unknown host";
    my $hispaddr = sockaddr_in($port, $hisiaddr);
    socket(SOCKET, PF_INET, SOCK_STREAM, $proto)   || die "socket: $!";
    connect(SOCKET, $hispaddr)          || die "bind: $!";
    my $rtime = '    ';
    read(SOCKET, $rtime, 4);
    close(SOCKET);
    my $histime = unpack("N", $rtime) - $SECS_of_70_YEARS ;
    printf "%8d %s\n", $histime - time, ctime($histime);
}
```

**Unix-Domain TCP Clients and Servers**

That's fine for Internet-domain clients and servers, but what about local communications? While you can use the same setup, sometimes you don't want to. Unix-domain sockets are local to the current host, and are often used internally to implement pipes. Unlike Internet domain sockets, Unix domain sockets can show up in the file system with an *ls*(1) listing.

```
% ls -l /dev/log
srw-rw-rw-  1 root            0 Oct 31 07:23 /dev/log
```

You can test for these with Perl's **–S** file test:

```
unless ( -S '/dev/log' ) {
    die "something's wicked with the print system";
}
```

Here's a sample Unix-domain client:

```
#!/usr/bin/perl -w
use Socket;
use strict;
my ($rendezvous, $line);

$rendezvous = shift || '/tmp/catsock';
socket(SOCK, PF_UNIX, SOCK_STREAM, 0)       || die "socket: $!";
connect(SOCK, sockaddr_un($rendezvous))     || die "connect: $!";
while (defined($line = <SOCK>)) {
    print $line;
}
exit;
```

And here's a corresponding server. You don't have to worry about silly network terminators here because Unix domain sockets are guaranteed to be on the localhost, and thus everything works right.

```
#!/usr/bin/perl -Tw
use strict;
use Socket;
use Carp;

BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }
```

```
my $NAME = '/tmp/catsock';
my $uaddr = sockaddr_un($NAME);
my $proto = getprotobyname('tcp');

socket(Server,PF_UNIX,SOCK_STREAM,0)        || die "socket: $!";
unlink($NAME);
bind  (Server, $uaddr)                      || die "bind: $!";
listen(Server,SOMAXCONN)                    || die "listen: $!";

logmsg "server started on $NAME";

my $waitedpid;

sub REAPER {
    $waitedpid = wait;
    $SIG{CHLD} = \&REAPER;  # loathe sysV
    logmsg "reaped $waitedpid" . ($? ? " with exit $?" : '');
}

$SIG{CHLD} = \&REAPER;

for ( $waitedpid = 0;
      accept(Client,Server) || $waitedpid;
      $waitedpid = 0, close Client)
{
    next if $waitedpid;
    logmsg "connection on $NAME";
    spawn sub {
        print "Hello there, it's now ", scalar localtime, "\n";
        exec '/usr/games/fortune' or die "can't exec fortune: $!";
    };
}
```

As you see, it's remarkably similar to the Internet domain TCP server, so much so, in fact, that we've omitted several duplicate functions--*spawn()*, *logmsg()*, *ctime()*, and *REAPER()*--which are exactly the same as in the other server.

So why would you ever want to use a Unix domain socket instead of a simpler named pipe? Because a named pipe doesn't give you sessions. You can't tell one process's data from another's. With socket programming, you get a separate session for each client: that's why *accept()* takes two arguments.

For example, let's say that you have a long running database server daemon that you want folks from the World Wide Web to be able to access, but only if they go through a CGI interface. You'd have a small, simple CGI program that does whatever checks and logging you feel like, and then acts as a Unix-domain client and connects to your private server.

**TCP Clients with IO::Socket**

For those preferring a higher-level interface to socket programming, the IO::Socket module provides an object-oriented approach. IO::Socket is included as part of the standard Perl distribution as of the 5.004 release. If you're running an earlier version of Perl, just fetch IO::Socket from CPAN, where you'll also find find modules providing easy interfaces to the following systems: DNS, FTP, Ident (RFC 931), NIS and NISPlus, NNTP, Ping, POP3, SMTP, SNMP, SSLeay, Telnet, and Time—just to name a few.

### A Simple Client

Here's a client that creates a TCP connection to the ‘‘daytime’’ service at port 13 of the host name ‘‘localhost’’ and prints out everything that the server there cares to provide.

```
#!/usr/bin/perl -w
use IO::Socket;
$remote = IO::Socket::INET->new(
                    Proto    => "tcp",
                    PeerAddr => "localhost",
                    PeerPort => "daytime(13)",
                )
              or die "cannot connect to daytime port at localhost";
while ( <$remote> ) { print }
```

When you run this program, you should get something back that looks like this:

```
Wed May 14 08:40:46 MDT 1997
```

Here are what those parameters to the `new` constructor mean:

`Proto`
: This is which protocol to use. In this case, the socket handle returned will be connected to a TCP socket, because we want a stream-oriented connection, that is, one that acts pretty much like a plain old file. Not all sockets are this of this type. For example, the UDP protocol can be used to make a datagram socket, used for message-passing.

`PeerAddr`
: This is the name or Internet address of the remote host the server is running on. We could have specified a longer name like `"www.perl.com"`, or an address like `"204.148.40.9"`. For demonstration purposes, we've used the special hostname `"localhost"`, which should always mean the current machine you're running on. The corresponding Internet address for localhost is `"127.1"`, if you'd rather use that.

`PeerPort`
: This is the service name or port number we'd like to connect to. We could have gotten away with using just `"daytime"` on systems with a well-configured system services file,[FOOTNOTE: The system services file is in */etc/services* under Unix] but just in case, we've specified the port number (13) in parentheses. Using just the number would also have worked, but constant numbers make careful programmers nervous.

Notice how the return value from the `new` constructor is used as a filehandle in the `while` loop? That's what's called an indirect filehandle, a scalar variable containing a filehandle. You can use it the same way you would a normal filehandle. For example, you can read one line from it this way:

```
$line = <$handle>;
```

all remaining lines from is this way:

```
@lines = <$handle>;
```

and send a line of data to it this way:

```
print $handle "some data\n";
```

### A Webget Client

Here's a simple client that takes a remote host to fetch a document from, and then a list of documents to get from that host. This is a more interesting client than the previous one because it first sends something to the server before fetching the server's response.

```
#!/usr/bin/perl -w
use IO::Socket;
unless (@ARGV > 1) { die "usage: $0 host document ..." }
$host = shift(@ARGV);
$EOL = "\015\012";
$BLANK = $EOL x 2;
foreach $document ( @ARGV ) {
    $remote = IO::Socket::INET->new( Proto     => "tcp",
                                     PeerAddr  => $host,
                                     PeerPort  => "http(80)",
                                    );
    unless ($remote) { die "cannot connect to http daemon on $host" }
    $remote->autoflush(1);
    print $remote "GET $document HTTP/1.0" . $BLANK;
    while ( <$remote> ) { print }
    close $remote;
}
```

The web server handing the "http" service, which is assumed to be at its standard port, number 80. If your the web server you're trying to connect to is at a different port (like 1080 or 8080), you should specify as the named-parameter pair, `PeerPort => 8080`. The `autoflush` method is used on the socket because otherwise the system would buffer up the output we sent it. (If you're on a Mac, you'll also need to change every `"\n"` in your code that sends data over the network to be a `"\015\012"` instead.)

Connecting to the server is only the first part of the process: once you have the connection, you have to use the server's language. Each server on the network has its own little command language that it expects as input. The string that we send to the server starting with "GET" is in HTTP syntax. In this case, we simply request each specified document. Yes, we really are making a new connection for each document, even though it's the same host. That's the way you always used to have to speak HTTP. Recent versions of web browsers may request that the remote server leave the connection open a little while, but the server doesn't have to honor such a request.

Here's an example of running that program, which we'll call *webget*:

```
% webget www.perl.com /guanaco.html
HTTP/1.1 404 File Not Found
Date: Thu, 08 May 1997 18:02:32 GMT
Server: Apache/1.2b6
Connection: close
Content-type: text/html

<HEAD><TITLE>404 File Not Found</TITLE></HEAD>
<BODY><H1>File Not Found</H1>
The requested URL /guanaco.html was not found on this server.<P>
</BODY>
```

Ok, so that's not very interesting, because it didn't find that particular document. But a long response wouldn't have fit on this page.

For a more fully-featured version of this program, you should look to the *lwp-request* program included with the LWP modules from CPAN.

**Interactive Client with IO::Socket**

Well, that's all fine if you want to send one command and get one answer, but what about setting up something fully interactive, somewhat like the way *telnet* works? That way you can type a line, get the answer, type a line, get the answer, etc.

This client is more complicated than the two we've done so far, but if you're on a system that supports the powerful `fork` call, the solution isn't that rough. Once you've made the connection to whatever service you'd like to chat with, call `fork` to clone your process. Each of these two identical process has a very simple job to do: the parent copies everything from the socket to standard output, while the child simultaneously copies everything from standard input to the socket. To accomplish the same thing using just one process would be *much* harder, because it's easier to code two processes to do one thing than it is to code one process to do two things. (This keep-it-simple principle a cornerstones of the Unix philosophy, and good software engineering as well, which is probably why it's spread to other systems.)

Here's the code:

```
#!/usr/bin/perl -w
use strict;
use IO::Socket;
my ($host, $port, $kidpid, $handle, $line);

unless (@ARGV == 2) { die "usage: $0 host port" }
($host, $port) = @ARGV;

# create a tcp connection to the specified host and port
$handle = IO::Socket::INET->new(Proto     => "tcp",
                                PeerAddr  => $host,
                                PeerPort  => $port)
       or die "can't connect to port $port on $host: $!";

$handle->autoflush(1);              # so output gets there right away
print STDERR "[Connected to $host:$port]\n";

# split the program into two processes, identical twins
die "can't fork: $!" unless defined($kidpid = fork());

# the if{} block runs only in the parent process
if ($kidpid) {
    # copy the socket to standard output
    while (defined ($line = <$handle>)) {
        print STDOUT $line;
    }
    kill("TERM", $kidpid);                  # send SIGTERM to child
}
# the else{} block runs only in the child process
else {
    # copy standard input to the socket
    while (defined ($line = <STDIN>)) {
        print $handle $line;
    }
}
```

The `kill` function in the parent's `if` block is there to send a signal to our child process (current running in the `else` block) as soon as the remote server has closed its end of the connection.

If the remote server sends data a byte at time, and you need that data immediately without waiting for a newline (which might not happen), you may wish to replace the `while` loop in the parent with the following:

```
my $byte;
while (sysread($handle, $byte, 1) == 1) {
    print STDOUT $byte;
}
```

Making a system call for each byte you want to read is not very efficient (to put it mildly) but is the simplest to explain and works reasonably well.

**TCP Servers with IO::Socket**

As always, setting up a server is little bit more involved than running a client. The model is that the server creates a special kind of socket that does nothing but listen on a particular port for incoming connections. It does this by calling the `IO::Socket::INET->new()` method with slightly different arguments than the client did.

Proto
: This is which protocol to use. Like our clients, we'll still specify `"tcp"` here.

LocalPort
: We specify a local port in the `LocalPort` argument, which we didn't do for the client. This is service name or port number for which you want to be the server. (Under Unix, ports under 1024 are restricted to the superuser.) In our sample, we'll use port 9000, but you can use any port that's not currently in use on your system. If you try to use one already in used, you'll get an "Address already in use" message. Under Unix, the `netstat -a` command will show which services current have servers.

Listen
: The `Listen` parameter is set to the maximum number of pending connections we can accept until we turn away incoming clients. Think of it as a call-waiting queue for your telephone. The low-level Socket module has a special symbol for the system maximum, which is SOMAXCONN.

Reuse
: The `Reuse` parameter is needed so that we restart our server manually without waiting a few minutes to allow system buffers to clear out.

Once the generic server socket has been created using the parameters listed above, the server then waits for a new client to connect to it. The server blocks in the `accept` method, which eventually an bidirectional connection to the remote client. (Make sure to autoflush this handle to circumvent buffering.)

To add to user-friendliness, our server prompts the user for commands. Most servers don't do this. Because of the prompt without a newline, you'll have to use the `sysread` variant of the interactive client above.

This server accepts one of five different commands, sending output back to the client. Note that unlike most network servers, this one only handles one incoming client at a time. Multithreaded servers are covered in Chapter 6 of the Camel as well as later in this manpage.

Here's the code. We'll

```
#!/usr/bin/perl -w
use IO::Socket;
use Net::hostent;              # for OO version of gethostbyaddr

$PORT = 9000;                  # pick something not in use
```

```
$server = IO::Socket::INET->new( Proto     => 'tcp',
                                 LocalPort => $PORT,
                                 Listen    => SOMAXCONN,
                                 Reuse     => 1);

die "can't setup server" unless $server;
print "[Server $0 accepting clients]\n";

while ($client = $server->accept()) {
  $client->autoflush(1);
  print $client "Welcome to $0; type help for command list.\n";
  $hostinfo = gethostbyaddr($client->peeraddr);
  printf "[Connect from %s]\n", $hostinfo->name || $client->peerhost;
  print $client "Command? ";
  while ( <$client>) {
    next unless /\S/;       # blank line
    if    (/quit|exit/i)    { last;                                     }
    elsif (/date|time/i)    { printf $client "%s\n", scalar localtime;  }
    elsif (/who/i )         { print  $client `who 2>&1`;                }
    elsif (/cookie/i )      { print  $client `/usr/games/fortune 2>&1`; }
    elsif (/motd/i )        { print  $client `cat /etc/motd 2>&1`;      }
    else {
      print $client "Commands: quit date who cookie motd\n";
    }
  } continue {
     print $client "Command? ";
  }
  close $client;
}
```

**UDP: Message Passing**

Another kind of client-server setup is one that uses not connections, but messages. UDP communications involve much lower overhead but also provide less reliability, as there are no promises that messages will arrive at all, let alone in order and unmangled. Still, UDP offers some advantages over TCP, including being able to "broadcast" or "multicast" to a whole bunch of destination hosts at once (usually on your local subnet). If you find yourself overly concerned about reliability and start building checks into your message system, then you probably should use just TCP to start with.

Here's a UDP program similar to the sample Internet TCP client given earlier. However, instead of checking one host at a time, the UDP version will check many of them asynchronously by simulating a multicast and then using *select()* to do a timed-out wait for I/O. To do something similar with TCP, you'd have to use a different socket handle for each host.

```
#!/usr/bin/perl -w
use strict;
use Socket;
use Sys::Hostname;

my ( $count, $hisiaddr, $hispaddr, $histime,
     $host, $iaddr, $paddr, $port, $proto,
     $rin, $rout, $rtime, $SECS_of_70_YEARS);

$SECS_of_70_YEARS      = 2208988800;
```

```
$iaddr = gethostbyname(hostname());
$proto = getprotobyname('udp');
$port = getservbyname('time', 'udp');
$paddr = sockaddr_in(0, $iaddr); # 0 means let kernel pick

socket(SOCKET, PF_INET, SOCK_DGRAM, $proto)   || die "socket: $!";
bind(SOCKET, $paddr)                          || die "bind: $!";

$| = 1;
printf "%-12s %8s %s\n",  "localhost", 0, scalar localtime time;
$count = 0;
for $host (@ARGV) {
    $count++;
    $hisiaddr = inet_aton($host)    || die "unknown host";
    $hispaddr = sockaddr_in($port, $hisiaddr);
    defined(send(SOCKET, 0, 0, $hispaddr))    || die "send $host: $!";
}

$rin = '';
vec($rin, fileno(SOCKET), 1) = 1;

# timeout after 10.0 seconds
while ($count && select($rout = $rin, undef, undef, 10.0)) {
    $rtime = '';
    ($hispaddr = recv(SOCKET, $rtime, 4, 0))        || die "recv: $!";
    ($port, $hisiaddr) = sockaddr_in($hispaddr);
    $host = gethostbyaddr($hisiaddr, AF_INET);
    $histime = unpack("N", $rtime) - $SECS_of_70_YEARS ;
    printf "%-12s ", $host;
    printf "%8d %s\n", $histime - time, scalar localtime($histime);
    $count--;
}
```

**SysV IPC**

While System V IPC isn't so widely used as sockets, it still has some interesting uses. You can't, however, effectively use SysV IPC or Berkeley *mmap()* to have shared memory so as to share a variable amongst several processes. That's because Perl would reallocate your string when you weren't wanting it to.

Here's a small example showing shared memory usage.

```
use IPC::SysV qw(IPC_PRIVATE IPC_RMID S_IRWXU S_IRWXG S_IRWXO);

$size = 2000;
$key = shmget(IPC_PRIVATE, $size, S_IRWXU|S_IRWXG|S_IRWXO) || die "$!";
print "shm key $key\n";

$message = "Message #1";
shmwrite($key, $message, 0, 60) || die "$!";
print "wrote: '$message'\n";
shmread($key, $buff, 0, 60) || die "$!";
print "read : '$buff'\n";
```

```
# the buffer of shmread is zero-character end-padded.
substr($buff, index($buff, "\0")) = '';
print "un" unless $buff eq $message;
print "swell\n";

print "deleting shm $key\n";
shmctl($key, IPC_RMID, 0) || die "$!";
```

Here's an example of a semaphore:

```
use IPC::SysV qw(IPC_CREAT);

$IPC_KEY = 1234;
$key = semget($IPC_KEY, 10, 0666 | IPC_CREAT ) || die "$!";
print "shm key $key\n";
```

Put this code in a separate file to be run in more than one process. Call the file *take*:

```
# create a semaphore

$IPC_KEY = 1234;
$key = semget($IPC_KEY,  0 , 0 );
die if !defined($key);

$semnum = 0;
$semflag = 0;

# 'take' semaphore
# wait for semaphore to be zero
$semop = 0;
$opstring1 = pack("sss", $semnum, $semop, $semflag);

# Increment the semaphore count
$semop = 1;
$opstring2 = pack("sss", $semnum, $semop,  $semflag);
$opstring = $opstring1 . $opstring2;

semop($key,$opstring) || die "$!";
```

Put this code in a separate file to be run in more than one process. Call this file *give*:

```
# 'give' the semaphore
# run this in the original process and you will see
# that the second process continues

$IPC_KEY = 1234;
$key = semget($IPC_KEY, 0, 0);
die if !defined($key);

$semnum = 0;
$semflag = 0;

# Decrement the semaphore count
$semop = -1;
$opstring = pack("sss", $semnum, $semop, $semflag);
```

```
semop($key,$opstring) || die "$!";
```

The SysV IPC code above was written long ago, and it's definitely clunky looking. For a more modern look, see the IPC::SysV module which is included with Perl starting from Perl 5.005.

**NOTES**

Most of these routines quietly but politely return `undef` when they fail instead of causing your program to die right then and there due to an uncaught exception. (Actually, some of the new *Socket* conversion functions *croak()* on bad arguments.) It is therefore essential to check return values from these functions. Always begin your socket programs this way for optimal success, and don't forget to add **−T** taint checking flag to the #! line for servers:

```
#!/usr/bin/perl -Tw
use strict;
use sigtrap;
use Socket;
```

**BUGS**

All these routines create system-specific portability problems. As noted elsewhere, Perl is at the mercy of your C libraries for much of its system behaviour. It's probably safest to assume broken SysV semantics for signals and to stick with simple TCP and UDP socket operations; e.g., don't try to pass open file descriptors over a local UDP datagram socket if you want your code to stand a chance of being portable.

As mentioned in the signals section, because few vendors provide C libraries that are safely re-entrant, the prudent programmer will do little else within a handler beyond setting a numeric variable that already exists; or, if locked into a slow (restarting) system call, using *die()* to raise an exception and *longjmp*(3) out. In fact, even these may in some cases cause a core dump. It's probably best to avoid signals except where they are absolutely inevitable. This will be addressed in a future release of Perl.

**AUTHOR**

Tom Christiansen, with occasional vestiges of Larry Wall's original version and suggestions from the Perl Porters.

**SEE ALSO**

There's a lot more to networking than this, but this should get you started.

For intrepid programmers, the indispensable textbook is *Unix Network Programming* by W. Richard Stevens (published by Addison-Wesley). Note that most books on networking address networking from the perspective of a C programmer; translation to Perl is left as an exercise for the reader.

The *IO::Socket*(3) manpage describes the object library, and the *Socket*(3) manpage describes the low-level interface to sockets. Besides the obvious functions in the *perlfunc* manpage, you should also check out the *modules* file at your nearest CPAN site. (See the *perlmodlib* manpage or best yet, the *Perl FAQ* for a description of what CPAN is and where to get it.)

Section 5 of the *modules* file is devoted to ''Networking, Device Control (modems), and Interprocess Communication'', and contains numerous unbundled modules numerous networking modules, Chat and Expect operations, CGI programming, DCE, FTP, IPC, NNTP, Proxy, Ptty, RPC, SNMP, SMTP, Telnet, Threads, and ToolTalk—just to name a few.

## NAME

perldebug – Perl debugging

## DESCRIPTION

First of all, have you tried using the **–w** switch?

### The Perl Debugger

"As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

*--Maurice Wilkes, 1949*

If you invoke Perl with the **–d** switch, your script runs under the Perl source debugger. This works like an interactive Perl environment, prompting for debugger commands that let you examine source code, set breakpoints, get stack backtraces, change the values of variables, etc. This is so convenient that you often fire up the debugger all by itself just to test out Perl constructs interactively to see what they do. For example:

```
perl -d -e 42
```

In Perl, the debugger is not a separate program as it usually is in the typical compiled environment. Instead, the **–d** flag tells the compiler to insert source information into the parse trees it's about to hand off to the interpreter. That means your code must first compile correctly for the debugger to work on it. Then when the interpreter starts up, it preloads a Perl library file containing the debugger itself.

The program will halt *right before* the first run-time executable statement (but see below regarding compile-time statements) and ask you to enter a debugger command. Contrary to popular expectations, whenever the debugger halts and shows you a line of code, it always displays the line it's *about* to execute, rather than the one it has just executed.

Any command not recognized by the debugger is directly executed (`eval`'d) as Perl code in the current package. (The debugger uses the DB package for its own state information.)

Leading white space before a command would cause the debugger to think it's *NOT* a debugger command but for Perl, so be careful not to do that.

#### Debugger Commands

The debugger understands the following commands:

h [command] Prints out a help message.

If you supply another debugger command as an argument to the `h` command, it prints out the description for just that command. The special argument of `h h` produces a more compact help listing, designed to fit together on one screen.

If the output of the `h` command (or any command, for that matter) scrolls past your screen, either precede the command with a leading pipe symbol so it's run through your pager, as in

```
DB> |h
```

You may change the pager which is used via `O pager=...` command.

p expr Same as `print {$DB::OUT} expr` in the current package. In particular, because this is just Perl's own **print** function, this means that nested data structures and objects are not dumped, unlike with the `x` command.

The `DB::OUT` filehandle is opened to */dev/tty*, regardless of where STDOUT may be redirected to.

x expr
: Evaluates its expression in list context and dumps out the result in a pretty-printed fashion. Nested data structures are printed out recursively, unlike the `print` function.

  The details of printout are governed by multiple `Options`.

V [pkg [vars]]
: Display all (or some) variables in package (defaulting to the `main` package) using a data pretty-printer (hashes show their keys and values so you see what's what, control characters are made printable, etc.). Make sure you don't put the type specifier (like `$`) there, just the symbol names, like this:

```
V DB filename line
```

  Use `~pattern` and `!pattern` for positive and negative regexps.

  Nested data structures are printed out in a legible fashion, unlike the `print` function.

  The details of printout are governed by multiple `Options`.

X [vars]
: Same as `V currentpackage [vars]`.

T
: Produce a stack backtrace. See below for details on its output.

s [expr]
: Single step. Executes until it reaches the beginning of another statement, descending into subroutine calls. If an expression is supplied that includes function calls, it too will be single-stepped.

n [expr]
: Next. Executes over subroutine calls, until it reaches the beginning of the next statement. If an expression is supplied that includes function calls, those functions will be executed with stops before each statement.

<CR>
: Repeat last `n` or `s` command.

c [line|sub]
: Continue, optionally inserting a one-time-only breakpoint at the specified line or subroutine.

l
: List next window of lines.

l min+incr
: List `incr+1` lines starting at `min`.

l min-max
: List lines `min` through `max`. `l -` is synonymous to `-`.

l line
: List a single line.

l subname
: List first window of lines from subroutine.

-
: List previous window of lines.

w [line]
: List window (a few lines) around the current line.

.
: Return debugger pointer to the last-executed line and print it out.

f filename
: Switch to viewing a different file or eval statement. If `filename` is not a full filename as found in values of `%INC`, it is considered as a regexp.

/pattern/
: Search forwards for pattern; final / is optional.

?pattern?
: Search backwards for pattern; final ? is optional.

L
: List all breakpoints and actions.

S [[!]pattern]
: List subroutine names [not] matching pattern.

t
: Toggle trace mode (see also `AutoTrace` Option).

t expr
: Trace through execution of expr. For example:

```
$ perl -de 42
Stack dump during die enabled outside of evals.
```

```
Loading DB routines from perl5db.pl patch level 0.94
Emacs support available.

Enter h or `h h' for help.

main::(-e:1):   0
  DB<1> sub foo { 14 }

  DB<2> sub bar { 3 }

  DB<3> t print foo() * bar()
main::((eval 172):3):   print foo() + bar();
main::foo((eval 168):2):
main::bar((eval 170):2):
42
```

or, with the `Option` `frame=2` set,

```
  DB<4> O f=2
             frame = '2'
  DB<5> t print foo() * bar()
3:      foo() * bar()
entering main::foo
 2:     sub foo { 14 };
exited main::foo
entering main::bar
 2:     sub bar { 3 };
exited main::bar
42
```

b [line] [condition]
: Set a breakpoint. If line is omitted, sets a breakpoint on the line that is about to be executed. If a condition is specified, it's evaluated each time the statement is reached and a breakpoint is taken only if the condition is true. Breakpoints may be set on only lines that begin an executable statement. Conditions don't use **if**:

```
b 237 $x > 30
b 237 ++$count237 < 11
b 33 /pattern/i
```

b subname [condition]
: Set a breakpoint at the first line of the named subroutine.

b postpone subname [condition]
: Set breakpoint at first line of subroutine after it is compiled.

b load filename
: Set breakpoint at the first executed line of the file. Filename should be a full name as found in values of `%INC`.

b compile subname
: Sets breakpoint at the first statement executed after the subroutine is compiled.

d [line]
: Delete a breakpoint at the specified line. If line is omitted, deletes the breakpoint on the line that is about to be executed.

D
: Delete all installed breakpoints.

a [line] command
: Set an action to be done before the line is executed. The sequence of steps taken by the debugger is

```
1. check for a breakpoint at this line
2. print the line if necessary (tracing)
3. do any actions associated with that line
4. prompt user if at a breakpoint or in single-step
5. evaluate line
```

For example, this will print out `$foo` every time line 53 is passed:

```
a 53 print "DB FOUND $foo\n"
```

A
: Delete all installed actions.

W [expr]
: Add a global watch-expression.

W
: Delete all watch-expressions.

O [opt[=val]] [op'val' [opt?]...
: Set or query values of options. val defaults to 1. opt can be abbreviated. Several options can be listed.

`recallCommand`, `ShellBang`
: The characters used to recall command or spawn shell. By default, these are both set to `!`.

`pager`
: Program to use for output of pager-piped commands (those beginning with a | character.) By default, `$ENV{PAGER}` will be used.

`tkRunning`
: Run Tk while prompting (with ReadLine).

`signalLevel`, `warnLevel`, `dieLevel`
: Level of verbosity. By default the debugger is in a sane verbose mode, thus it will print backtraces on all the warnings and die-messages which are going to be printed out, and will print a message when interesting uncaught signals arrive.

To disable this behaviour, set these values to 0. If `dieLevel` is 2, then the messages which will be caught by surrounding `eval` are also printed.

`AutoTrace`
: Trace mode (similar to `t` command, but can be put into `PERLDB_OPTS`).

`LineInfo`
: File or pipe to print line number info to. If it is a pipe (say, `|visual_perl_db`), then a short, "emacs like" message is used.

`inhibit_exit`
: If 0, allows *stepping off* the end of the script.

`PrintRet`
: affects printing of return value after `r` command.

`ornaments`
: affects screen appearance of the command line (see the *Term::ReadLine* manpage).

`frame`
: affects printing messages on entry and exit from subroutines. If `frame & 2` is false, messages are printed on entry only. (Printing on exit may be useful if *inter* (di)spersed with other messages.)

If `frame & 4`, arguments to functions are printed as well as the context and caller info. If `frame & 8`, overloaded `stringify` and `tied FETCH` are enabled on the printed arguments. If `frame & 16`, the return value from the subroutine is printed as well.

The length at which the argument list is truncated is governed by the next option:

`maxTraceLen`
: length at which the argument list is truncated when `frame` option's bit 4 is set.

The following options affect what happens with `V`, `X`, and `x` commands:

`arrayDepth`, `hashDepth`
: Print only first N elements ('' for all).

`compactDump`, `veryCompact`
: Change style of array and hash dump. If `compactDump`, short array may be printed on one line.

`globPrint`
: Whether to print contents of globs.

`DumpDBFiles`
: Dump arrays holding debugged files.

`DumpPackages`
: Dump symbol tables of packages.

`DumpReused`
: Dump contents of "reused" addresses.

`quote`, `HighBit`, `undefPrint`
: Change style of string dump. Default value of `quote` is `auto`, one can enable either double-quotish dump, or single-quotish by setting it to `"` or `'`. By default, characters with high bit set are printed *as is*.

`UsageOnly`
: *very* rudimentally per-package memory usage dump. Calculates total size of strings in variables in the package.

During startup options are initialized from `$ENV{PERLDB_OPTS}`. You can put additional initialization options `TTY`, `noTTY`, `ReadLine`, and `NonStop` there.

Example rc file:

```
&parse_options("NonStop=1 LineInfo=db.out AutoTrace");
```

The script will run without human intervention, putting trace information into the file *db.out*. (If you interrupt it, you would better reset `LineInfo` to something "interactive"!)

`TTY`
: The TTY to use for debugging I/O.

`noTTY`
: If set, goes in `NonStop` mode, and would not connect to a TTY. If interrupt (or if control goes to debugger via explicit setting of `$DB::signal` or `$DB::single` from the Perl script), connects to a TTY specified by the `TTY` option at startup, or to a TTY found at runtime using `Term::Rendezvous` module of your choice.

This module should implement a method `new` which returns an object with two methods: `IN` and `OUT`, returning two filehandles to use for debugging input and output correspondingly. Method `new` may inspect an argument which is a value of `$ENV{PERLDB_NOTTY}` at startup, or is `"/tmp/perldbtty$$"` otherwise.

`ReadLine`
: If false, readline support in debugger is disabled, so you can debug ReadLine applications.

`NonStop`
: If set, debugger goes into noninteractive mode until interrupted, or programmatically by setting `$DB::signal` or `$DB::single`.

Here's an example of using the `$ENV{PERLDB_OPTS}` variable:

```
$ PERLDB_OPTS="N f=2" perl -d myprogram
```

will run the script `myprogram` without human intervention, printing out the call tree with entry and exit points. Note that `N f=2` is equivalent to `NonStop=1 frame=2`. Note also that at the moment when this documentation was written all the options to the debugger could be uniquely abbreviated by the first letter (with exception of `Dump*` options).

Other examples may include

```
$ PERLDB_OPTS="N f A L=listing" perl -d myprogram
```

- runs script noninteractively, printing info on each entry into a subroutine and each executed line into the file *listing*. (If you interrupt it, you would better reset `LineInfo` to something "interactive"!)

```
$ env "PERLDB_OPTS=R=0 TTY=/dev/ttyc" perl -d myprogram
```

may be useful for debugging a program which uses `Term::ReadLine` itself. Do not forget detach shell from the TTY in the window which corresponds to */dev/ttyc*, say, by issuing a command like

```
$ sleep 1000000
```

See the section on *Debugger Internals* below for more details.

< [ command ]
: Set an action (Perl command) to happen before every debugger prompt. A multi-line command may be entered by backslashing the newlines. If `command` is missing, resets the list of actions.

<< command
: Add an action (Perl command) to happen before every debugger prompt. A multi-line command may be entered by backslashing the newlines.

> command
: Set an action (Perl command) to happen after the prompt when you've just given a command to return to executing the script. A multi-line command may be entered by backslashing the newlines. If `command` is missing, resets the list of actions.

>> command
: Adds an action (Perl command) to happen after the prompt when you've just given a command to return to executing the script. A multi-line command may be entered by backslashing the newlines.

{ [ command ]
: Set an action (debugger command) to happen before every debugger prompt. A multi-line command may be entered by backslashing the newlines. If `command` is missing, resets the list of actions.

{{ command
: Add an action (debugger command) to happen before every debugger prompt. A multi-line command may be entered by backslashing the newlines.

! number
: Redo a previous command (default previous command).

! –number
: Redo number'th-to-last command.

! pattern
: Redo last command that started with pattern. See `O recallCommand`, too.

!! cmd
: Run cmd in a subprocess (reads from DB::IN, writes to DB::OUT) See `O shellBang` too.

H –number
: Display last n commands. Only commands longer than one character are listed. If number is omitted, lists them all.

q or ^D
: Quit. ("quit'' doesn't work for this.) This is the only supported way to exit the debugger, though typing `exit` twice may do it too.

  Set an `O`ption `inhibit_exit` to 0 if you want to be able to *step off* the end the script. You may also need to set `$finished` to 0 at some moment if you want to step through global destruction.

R
: Restart the debugger by **exec**ing a new session. It tries to maintain your history across this, but internal settings and command line options may be lost.

  Currently the following setting are preserved: history, breakpoints, actions, debugger `O`ptions, and the following command line options: **−w**, **−I**, and **−e**.

|dbcmd
: Run debugger command, piping DB::OUT to current pager.

||dbcmd
: Same as `|dbcmd` but DB::OUT is temporarily **select**ed as well. Often used with commands that would otherwise produce long output, such as

```
|V main
```

= [alias value]
: Define a command alias, like

```
= quit q
```

  or list current aliases.

command
: Execute command as a Perl statement. A missing semicolon will be supplied.

m expr
: The expression is evaluated, and the methods which may be applied to the result are listed.

m package
: The methods which may be applied to objects in the `package` are listed.

**Debugger input/output**

Prompt
: The debugger prompt is something like

```
DB<8>
```

  or even

```
DB<<17>>
```

  where that number is the command number, which you'd use to access with the builtin **csh**−like history mechanism, e.g., `!17` would repeat command number 17. The number of angle brackets indicates the depth of the debugger. You could get more than one set of brackets, for example, if you'd already at a breakpoint and then printed out the result of a function call that itself also has a breakpoint, or you step into an expression via `s/n/t expression` command.

Multiline commands
: If you want to enter a multi-line command, such as a subroutine definition with several statements, or a format, you may escape the newline that would normally end the debugger command with a backslash. Here's an example:

```
DB<1> for (1..4) {         \
cont:     print "ok\n";   \
cont: }
ok
ok
ok
ok
```

Note that this business of escaping a newline is specific to interactive commands typed into the debugger.

Stack backtrace

Here's an example of what a stack backtrace via `T` command might look like:

```
$ = main::infested called from file `Ambulation.pm' line 10
@ = Ambulation::legs(1, 2, 3, 4) called from file `camel_flea' line 7
$ = main::pests('bactrian', 4) called from file `camel_flea' line 4
```

The left-hand character up there tells whether the function was called in a scalar or list context (we bet you can tell which is which). What that says is that you were in the function `main::infested` when you ran the stack dump, and that it was called in a scalar context from line 10 of the file *Ambulation.pm*, but without any arguments at all, meaning it was called as `&infested`. The next stack frame shows that the function `Ambulation::legs` was called in a list context from the *camel_flea* file with four arguments. The last stack frame shows that `main::pests` was called in a scalar context, also from *camel_flea*, but from line 4.

Note that if you execute `T` command from inside an active `use` statement, the backtrace will contain both `require` frame and an `eval`) frame.

Listing

Listing given via different flavors of `l` command looks like this:

```
  DB<<13>> l
101:                @i{@i} = ();
102:b               @isa{@i,$pack} = ()
103                     if(exists $i{$prevpack} || exists $isa{$pack});
104             }
105
106             next
107==>              if(exists $isa{$pack});
108
109:a            if ($extra-- > 0) {
110:                %isa = ($pack,1);
```

Note that the breakable lines are marked with `:`, lines with breakpoints are marked by `b`, with actions by `a`, and the next executed line is marked by `==>`.

Frame listing

When `frame` option is set, debugger would print entered (and optionally exited) subroutines in different styles.

What follows is the start of the listing of

```
env "PERLDB_OPTS=f=n N" perl -d -V
```

for different values of `n`:

1

```
entering main::BEGIN
 entering Config::BEGIN
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 Package lib/Config.pm.
 entering Config::TIEHASH
 entering Exporter::import
  entering Exporter::export
entering Config::myconfig
 entering Config::FETCH
 entering Config::FETCH
 entering Config::FETCH
 entering Config::FETCH
```

2

```
entering main::BEGIN
 entering Config::BEGIN
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 exited Config::BEGIN
 Package lib/Config.pm.
 entering Config::TIEHASH
 exited Config::TIEHASH
 entering Exporter::import
  entering Exporter::export
  exited Exporter::export
 exited Exporter::import
exited main::BEGIN
entering Config::myconfig
 entering Config::FETCH
 exited Config::FETCH
 entering Config::FETCH
 exited Config::FETCH
 entering Config::FETCH
```

4

```
in  $=main::BEGIN() from /dev/nul:0
 in  $=Config::BEGIN() from lib/Config.pm:2
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 Package lib/Config.pm.
 in  $=Config::TIEHASH('Config') from lib/Config.pm:644
 in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/n
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') fr
in  @=Config::myconfig() from /dev/nul:0
 in  $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'PATCHLEVEL') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'SUBVERSION') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'osname') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'osvers') from lib/Config.pm:574
```

6

```
in  $=main::BEGIN() from /dev/nul:0
 in  $=Config::BEGIN() from lib/Config.pm:2
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 out $=Config::BEGIN() from lib/Config.pm:0
 Package lib/Config.pm.
 in  $=Config::TIEHASH('Config') from lib/Config.pm:644
 out $=Config::TIEHASH('Config') from lib/Config.pm:644
 in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/n
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') fr
  out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') fr
 out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/n
out $=main::BEGIN() from /dev/nul:0
in  @=Config::myconfig() from /dev/nul:0
 in  $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'PATCHLEVEL') from lib/Config.pm:574
 out $=Config::FETCH(ref(Config), 'PATCHLEVEL') from lib/Config.pm:574
 in  $=Config::FETCH(ref(Config), 'SUBVERSION') from lib/Config.pm:574
```

14

```
in  $=main::BEGIN() from /dev/nul:0
 in  $=Config::BEGIN() from lib/Config.pm:2
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
 out $=Config::BEGIN() from lib/Config.pm:0
 Package lib/Config.pm.
 in  $=Config::TIEHASH('Config') from lib/Config.pm:644
 out $=Config::TIEHASH('Config') from lib/Config.pm:644
 in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/n
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') fr
  out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') fr
 out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/n
out $=main::BEGIN() from /dev/nul:0
in  @=Config::myconfig() from /dev/nul:0
 in  $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from lib/Config
 out $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from lib/Config
 in  $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from lib/Config
 out $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from lib/Config
```

30

```
in  $=CODE(0x15eca4)() from /dev/null:0
 in  $=CODE(0x182528)() from lib/Config.pm:2
  Package lib/Exporter.pm.
 out $=CODE(0x182528)() from lib/Config.pm:0
 scalar context return from CODE(0x182528): undef
 Package lib/Config.pm.
 in  $=Config::TIEHASH('Config') from lib/Config.pm:628
 out $=Config::TIEHASH('Config') from lib/Config.pm:628
 scalar context return from Config::TIEHASH:   empty hash
 in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/n
  in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') fr
  out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') fr
  scalar context return from Exporter::export: ''
 out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/n
 scalar context return from Exporter::import: ''
```

In all the cases indentation of lines shows the call tree, if bit 2 of `frame` is set, then a line is printed on exit from a subroutine as well, if bit 4 is set, then the arguments are printed as well as the caller info, if bit 8 is set, the arguments are printed even if they are tied or references, if bit 16 is set, the return value is printed as well.

When a package is compiled, a line like this

```
Package lib/Carp.pm.
```

is printed with proper indentation.

### Debugging compile-time statements

If you have any compile-time executable statements (code within a BEGIN block or a `use` statement), these will `NOT` be stopped by debugger, although `requires` will (and compile-time statements can be traced with `AutoTrace` option set in `PERLDB_OPTS`). From your own Perl code, however, you can transfer control back to the debugger using the following statement, which is harmless if the debugger is not running:

```
$DB::single = 1;
```

If you set `$DB::single` to the value 2, it's equivalent to having just typed the `n` command, whereas a value of 1 means the `s` command. The `$DB::trace` variable should be set to 1 to simulate having typed the `t` command.

Another way to debug compile-time code is to start debugger, set a breakpoint on *load* of some module thusly

```
  DB<7> b load f:/perllib/lib/Carp.pm
Will stop on load of `f:/perllib/lib/Carp.pm'.
```

and restart debugger by `R` command (if possible). One can use `b compile subname` for the same purpose.

### Debugger Customization

Most probably you do not want to modify the debugger, it contains enough hooks to satisfy most needs. You may change the behaviour of debugger from the debugger itself, using `Options`, from the command line via `PERLDB_OPTS` environment variable, and from *customization files*.

You can do some customization by setting up a *.perldb* file which contains initialization code. For instance, you could make aliases like these (the last one is one people expect to be there):

```
$DB::alias{'len'}  = 's/^len(.*)/p length($1)/';
$DB::alias{'stop'} = 's/^stop (at|in)/b/';
$DB::alias{'ps'}   = 's/^ps\b/p scalar /';
$DB::alias{'quit'} = 's/^quit(\s*)/exit\$/';
```

One changes options from *.perldb* file via calls like this one;

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace=1 frame=2");
```

(the code is executed in the package `DB`). Note that *.perldb* is processed before processing `PERLDB_OPTS`. If *.perldb* defines the subroutine `afterinit`, it is called after all the debugger initialization ends. *.perldb* may be contained in the current directory, or in the `LOGDIR`/`HOME` directory.

If you want to modify the debugger, copy *perl5db.pl* from the Perl library to another name and modify it as necessary. You'll also want to set your `PERL5DB` environment variable to say something like this:

```
BEGIN { require "myperl5db.pl" }
```

As the last resort, one can use `PERL5DB` to customize debugger by directly setting internal variables or calling debugger functions.

### Readline Support

As shipped, the only command line history supplied is a simplistic one that checks for leading exclamation points. However, if you install the Term::ReadKey and Term::ReadLine modules from CPAN, you will have full editing capabilities much like GNU *readline*(3) provides. Look for these in the *modules/by-module/Term* directory on CPAN.

A rudimentary command line completion is also available. Unfortunately, the names of lexical variables are not available for completion.

### Editor Support for Debugging

If you have GNU **emacs** installed on your system, it can interact with the Perl debugger to provide an integrated software development environment reminiscent of its interactions with C debuggers.

Perl is also delivered with a start file for making **emacs** act like a syntax-directed editor that understands (some of) Perl's syntax. Look in the *emacs* directory of the Perl source distribution.

(Historically, a similar setup for interacting with **vi** and the X11 window system had also been available, but at the time of this writing, no debugger support for **vi** currently exists.)

### The Perl Profiler

If you wish to supply an alternative debugger for Perl to run, just invoke your script with a colon and a package argument given to the **–d** flag. One of the most popular alternative debuggers for Perl is **DProf**, the Perl profiler. As of this writing, **DProf** is not included with the standard Perl distribution, but it is expected to be included soon, for certain values of "soon".

Meanwhile, you can fetch the Devel::Dprof module from CPAN. Assuming it's properly installed on your system, to profile your Perl program in the file *mycode.pl*, just type:

```
perl -d:DProf mycode.pl
```

When the script terminates the profiler will dump the profile information to a file called *tmon.out*. A tool like **dprofpp** (also supplied with the Devel::DProf package) can be used to interpret the information which is in that profile.

### Debugger support in perl

When you call the **caller** function (see the `caller` entry in the *perlfunc* manpage) from the package `DB`, Perl sets the array `@DB::args` to contain the arguments the corresponding stack frame was called with.

If perl is run with **–d** option, the following additional features are enabled (cf. the section on *$^P* in the *perlvar* manpage):

- Perl inserts the contents of `$ENV{PERL5DB}` (or `BEGIN {require 'perl5db.pl'}` if not present) before the first line of the application.
- The array `@{"_<$filename"}` is the line-by-line contents of `$filename` for all the compiled files. Same for `eval`ed strings which contain subroutines, or which are currently executed. The `$filename` for `eval`ed strings looks like `(eval 34)`.
- The hash `%{"_<$filename"}` contains breakpoints and action (it is keyed by line number), and individual entries are settable (as opposed to the whole hash). Only true/false is important to Perl, though the values used by *perl5db.pl* have the form `"$break_condition\0$action"`. Values are magical in numeric context: they are zeros if the line is not breakable.

  Same for evaluated strings which contain subroutines, or which are currently executed. The `$filename` for `eval`ed strings looks like `(eval 34)`.
- The scalar `${"_<$filename"}` contains `"_<$filename"`. Same for evaluated strings which contain subroutines, or which are currently executed. The `$filename` for `eval`ed strings looks like `(eval 34)`.
- After each `require`d file is compiled, but before it is executed, `DB::postponed(*{"_<$filename"})` is called (if subroutine `DB::postponed` exists). Here the `$filename` is the expanded name of the `require`d file (as found in values of `%INC`).
- After each subroutine `subname` is compiled existence of `$DB::postponed{subname}` is checked. If this key exists, `DB::postponed(subname)` is called (if subroutine `DB::postponed` exists).
- A hash `%DB::sub` is maintained, with keys being subroutine names, values having the form `filename:startline-endline`. `filename` has the form `(eval 31)` for subroutines defined inside `eval`s.
- When execution of the application reaches a place that can have a breakpoint, a call to `DB::DB()` is performed if any one of variables `$DB::trace`, `$DB::single`, or `$DB::signal` is true. (Note that these variables are not `local`izable.) This feature is disabled when the control is inside `DB::DB()` or functions called from it (unless `$^D & (1<<30)`).
- When execution of the application reaches a subroutine call, a call to `&DB::sub`(*args*) is performed instead, with `$DB::sub` being the name of the called subroutine. (Unless the subroutine is compiled in the package `DB`.)

Note that if `&DB::sub` needs some external data to be setup for it to work, no subroutine call is possible until this is done. For the standard debugger `$DB::deep` (how many levels of recursion deep into the debugger you can go before a mandatory break) gives an example of such a dependency.

The minimal working debugger consists of one line

```
sub DB::DB {}
```

which is quite handy as contents of `PERL5DB` environment variable:

```
env "PERL5DB=sub DB::DB {}" perl -d your-script
```

Another (a little bit more useful) minimal debugger can be created with the only line being

```
sub DB::DB {print ++$i; scalar <STDIN>}
```

This debugger would print the sequential number of encountered statement, and would wait for your `CR` to continue.

The following debugger is quite functional:

```
{
  package DB;
  sub DB  {}
  sub sub {print ++$i, " $sub\n"; &$sub}
}
```

It prints the sequential number of subroutine call and the name of the called subroutine. Note that `&DB::sub` should be compiled into the package `DB`.

### Debugger Internals

At the start, the debugger reads your rc file (*./.perldb* or *~/.perldb* under Unix), which can set important options. This file may define a subroutine `&afterinit` to be executed after the debugger is initialized.

After the rc file is read, the debugger reads environment variable PERLDB_OPTS and parses it as a rest of `O ...` line in debugger prompt.

It also maintains magical internal variables, such as `@DB::dbline`, `%DB::dbline`, which are aliases for `@{"::_<current_file"} %{"::_<current_file"}`. Here `current_file` is the currently selected (with the debugger's `f` command, or by flow of execution) file.

Some functions are provided to simplify customization. See the section on *Debugger Customization* for description of `DB::parse_options(string)`. The function `DB::dump_trace(skip[, count])` skips the specified number of frames, and returns a list containing info about the caller frames (all if `count` is missing). Each entry is a hash with keys `context` (`$` or `@`), `sub` (subroutine name, or info about eval), `args` (`undef` or a reference to an array), `file`, and `line`.

The function `DB::print_trace(FH, skip[, count[, short]])` prints formatted info about caller frames. The last two functions may be convenient as arguments to <, << commands.

### Other resources

You did try the **–w** switch, didn't you?

#### BUGS

You cannot get the stack frame information or otherwise debug functions that were not compiled by Perl, such as C or C++ extensions.

If you alter your `@_` arguments in a subroutine (such as with **shift** or **pop**, the stack backtrace will not show the original values.

## Debugging Perl memory usage

Perl is *very* frivolous with memory. There is a saying that to estimate memory usage of Perl, assume a reasonable algorithm of allocation, and multiply your estimages by 10. This is not absolutely true, but may give you a good grasp of what happens.

Say, an integer cannot take less than 20 bytes of memory, a float cannot take less than 24 bytes, a string cannot take less than 32 bytes (all these examples assume 32-bit architectures, the result are much worse on 64-bit architectures). If a variable is accessed in two of three different ways (which require an integer, a float, or a string), the memory footprint may increase by another 20 bytes. A sloppy *malloc()* implementation will make these numbers yet more.

On the opposite end of the scale, a declaration like

```
sub foo;
```

may take (on some versions of perl) up to 500 bytes of memory.

Off-the-cuff anecdotal estimates of a code bloat give a factor around 8. This means that the compiled form of reasonable (commented indented etc.) code will take approximately 8 times more than the disk space the code takes.

There are two Perl-specific ways to analyze the memory usage: `$ENV{PERL_DEBUG_MSTATS}` and **-DL** switch. First one is available only if perl is compiled with Perl's *malloc()*, the second one only if Perl compiled with `-DDEBUGGING` (as with giving `-D optimise=-g` option to *Configure*).

### Using `$ENV{PERL_DEBUG_MSTATS}`

If your perl is using Perl's *malloc()*, and compiled with correct switches (this is the default), then it will print memory usage statistics after compiling your code (if `$ENV{PERL_DEBUG_MSTATS} > 1`), and before termination of the script (if `$ENV{PERL_DEBUG_MSTATS} >= 1`). The report format is similar to one in the following example:

```
env PERL_DEBUG_MSTATS=2 perl -e "require Carp"
Memory allocation statistics after compilation: (buckets 4(4)..8188(8192)
   14216 free:   130   117    28     7     9   0   2     2   1 0 0
              437    61    36     0     5
   60924 used:   125   137   161    55     7   8   6    16   2 0 1
               74   109   304    84    20
Total sbrk(): 77824/21:119. Odd ends: pad+heads+chain+tail: 0+636+0+2048.
Memory allocation statistics after execution:   (buckets 4(4)..8188(8192)
   30888 free:   245    78    85    13     6   2   1     3   2 0 1
              315   162    39    42    11
  175816 used:   265   176  1112   111    26  22  11    27   2 1 1
              196   178  1066   798    39
Total sbrk(): 215040/47:145. Odd ends: pad+heads+chain+tail: 0+2192+0+6144.
```

It is possible to ask for such a statistic at arbitrary moment by usind *Devel::Peek::mstats()* (module Devel::Peek is available on CPAN).

Here is the explanation of different parts of the format:

`buckets SMALLEST(APPROX)..GREATEST(APPROX)`

Perl's *malloc()* uses bucketed allocations. Every request is rounded up to the closest bucket size available, and a bucket of these size is taken from the pool of the buckets of this size.

The above line describes limits of buckets currently in use. Each bucket has two sizes: memory footprint, and the maximal size of user data which may be put into this bucket. Say, in the above example the smallest bucket is both sizes 4. The biggest bucket has usable size 8188, and the memory footprint 8192.

With debugging Perl some buckets may have negative usable size. This means that these buckets cannot (and will not) be used. For greater buckets the memory footprint may be one page greater than a power of 2. In such a case the corresponding power of two is printed instead in the `APPROX` field above.

Free/Used

The following 1 or 2 rows of numbers correspond to the number of buckets of each size between `SMALLEST` and `GREATEST`. In the first row the sizes (memory footprints) of buckets are powers of two (or possibly one page greater). In the second row (if present) the memory footprints of the buckets are between memory footprints of two buckets "above".

Say, with the above example the memory footprints are (with current algorith)

```
free:    8     16    32    64    128  256 512 1024 2048 4096 8192
      4     12    24    48    80
```

With non-`DEBUGGING` perl the buckets starting from `128`-long ones have 4-byte overhead, thus 8192-long bucket may take up to 8188-byte-long allocations.

`Total sbrk(): SBRKed/SBRKs:CONTINUOUS`

The first two fields give the total amount of memory perl *sbrk()*ed, and number of *sbrk()*s used. The third number is what perl thinks about continuity of returned chunks. As far as this number is positive, *malloc()* will assume that it is probable that *sbrk()* will provide continuous memory.

The amounts *sbrk()*ed by external libraries is not counted.

`pad: 0`

The amount of *sbrk()*ed memory needed to keep buckets aligned.

`heads: 2192`

While memory overhead of bigger buckets is kept inside the bucket, for smaller buckets it is kept in separate areas. This field gives the total size of these areas.

`chain: 0`

*malloc()* may want to subdivide a bigger bucket into smaller buckets. If only a part of the deceased-bucket is left non-subdivided, the rest is kept as an element of a linked list. This field gives the total size of these chunks.

`tail: 6144`

To minimize amount of *sbrk()*s *malloc()* asks for more memory. This field gives the size of the yet-unused part, which is *sbrk()*ed, but never touched.

**Example of using –DL** switch

Below we show how to analyse memory usage by

```
do 'lib/auto/POSIX/autosplit.ix';
```

The file in question contains a header and 146 lines similar to

```
sub getcwd ;
```

**Note:** *the discussion below supposes 32-bit architecture. In the newer versions of perl the memory usage of the constructs discussed here is much improved, but the story discussed below is a real-life story. This story is very terse, and assumes more than cursory knowledge of Perl internals.*

Here is the itemized list of Perl allocations performed during parsing of this file:

```
!!! "after" at test.pl line 3.
   Id  subtot   4   8  12  16  20  24  28  32  36  40  48  56  64  72  80 80+
 0 02   13752   .   .   .   . 294   .   .   .   .   .   .   .   .   .   .   4
 0 54    5545   .   .   8 124  16   .   .   .   1   1   .   .   .   .   .   3
 5 05      32   .   .   .   .   .   .   .   1   .   .   .   .   .   .   .   .
 6 02    7152   .   .   .   .   .   .   .   .   .   . 149   .   .   .   .   .
 7 02    3600   .   .   .   .   . 150   .   .   .   .   .   .   .   .   .   .
 7 03      64   .  -1   .   1   .   .   2   .   .   .   .   .   .   .   .   .
 7 04    7056   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   7
 7 17   38404   .   .   .   .   .   .   .   1   .   . 442 149   .   . 147   .
 9 03    2078  17 249  32   .   .   .   .   2   .   .   .   .   .   .   .   .
```

To see this list insert two `warn('!...')` statements around the call:

```
warn('!');
do 'lib/auto/POSIX/autosplit.ix';
warn('!!! "after"');
```

and run it with **–DL** option. The first *warn()* will print memory allocation info before the parsing of the file, and will memorize the statistics at this point (we ignore what it prints). The second *warn()* will print increments w.r.t. this memorized statistics. This is the above printout.

Different *Id*s on the left correspond to different subsystems of perl interpreter, they are just first argument given to perl memory allocation API *New()*. To find what `9 03` means `grep` the perl source for `903`. You will see that it is *util.c*, function *savepvn()*. This function is used to store a copy of existing chunk of memory. Using C debugger, one can see that it is called either directly from *gv_init()*, or via *sv_magic()*, and *gv_init()* is called from *gv_fetchpv()* – which is called from *newSUB()*.

**Note:** to reach this place in debugger and skip all the calls to savepvn during the compilation of the main script, set a C breakpoint in *Perl_warn()*, `continue` this point is reached, *then* set breakpoint in *Perl_savepvn()*. Note that you may need to skip a handful of *Perl_savepvn()* which do not correspond to mass production of CVs (there are more `903` allocations than 146 similar lines of *lib/auto/POSIX/autosplit.ix*). Note also that `Perl_` prefixes are added by macroization code in perl header files to avoid conflicts with external libraries.

Anyway, we see that `903` ids correspond to creation of globs, twice per glob – for glob name, and glob stringification magic.

Here are explanations for other *Id*s above:

`717` is for creation of bigger `XPV*` structures. In the above case it creates 3 `AV` per subroutine, one for a list of lexical variable names, one for a scratchpad (which contains lexical variables and `targets`), and one for the array of scratchpads needed for recursion.

It also creates a `GV` and a `CV` per subroutine (all called from *start_subparse()*).

`002` Creates C array corresponding to the `AV` of scratchpads, and the scratchpad itself (the first fake entry of this scratchpad is created though the subroutine itself is not defined yet).

It also creates C arrays to keep data for the stash (this is one `HV`, but it grows, thus there are 4 big allocations: the big chunks are not freeed, but are kept as additional arenas for `SV` allocations).

`054` creates a `HEK` for the name of the glob for the subroutine (this name is a key in a *stash*).

Big allocations with this *Id* correspond to allocations of new arenas to keep `HE`.

`602` creates a `GP` for the glob for the subroutine.

`702` creates the `MAGIC` for the glob for the subroutine.

`704` creates *arenas* which keep SVs.

**–DL** details

If Perl is run with **–DL** option, then *warn()*s which start with '!' behave specially. They print a list of *categories* of memory allocations, and statistics of allocations of different sizes for these categories.

If *warn()* string starts with

`!!!` print changed categories only, print the differences in counts of allocations;

`!!` print grown categories only; print the absolute values of counts, and totals;

`!` print nonempty categories, print the absolute values of counts and totals.

### Limitations of –DL statistic

If an extension or an external library does not use Perl API to allocate memory, these allocations are not counted.

## Debugging regular expressions

There are two ways to enable debugging output for regular expressions.

If your perl is compiled with `-DDEBUGGING`, you may use the **–Dr** flag on the command line.

Otherwise, one can `use re 'debug'`, which has effects both at compile time, and at run time (and is *not* lexically scoped).

### Compile-time output

The debugging output for the compile time looks like this:

```
compiling RE `[bc]d(ef*g)+h[ij]k$'
size 43 first at 1
   1: ANYOF(11)
  11: EXACT <d>(13)
  13: CURLYX {1,32767}(27)
  15:   OPEN1(17)
  17:     EXACT <e>(19)
  19:     STAR(22)
  20:       EXACT <f>(0)
  22:     EXACT <g>(24)
  24:   CLOSE1(26)
  26:   WHILEM(0)
  27: NOTHING(28)
  28: EXACT <h>(30)
  30: ANYOF(40)
  40: EXACT <k>(42)
  42: EOL(43)
  43: END(0)
anchored `de' at 1 floating `gh' at 3..2147483647 (checking floating)
                                  stclass `ANYOF' minlen 7
```

The first line shows the pre-compiled form of the regexp, and the second shows the size of the compiled form (in arbitrary units, usually 4-byte words) and the label *id* of the first node which does a match.

The last line (split into two lines in the above) contains the optimizer info. In the example shown, the optimizer found that the match should contain a substring `de` at the offset 1, and substring `gh` at some offset between 3 and infinity. Moreover, when checking for these substrings (to abandon impossible matches quickly) it will check for the substring `gh` before checking for the substring `de`. The optimizer may also use the knowledge that the match starts (at the `first` *id*) with a character class, and the match cannot be shorter than 7 chars.

The fields of interest which may appear in the last line are

`anchored` *STRING* `at` *POS*

`floating` *STRING* `at` *POS1..POS2*
: see above;

`matching floating/anchored`
: which substring to check first;

`minlen`
: the minimal length of the match;

`stclass` *TYPE*
: The type of the first matching node.

`noscan`
: which advises to not scan for the found substrings;

`isall`
: which says that the optimizer info is in fact all that the regular expression contains (thus one does not need to enter the RE engine at all);

`GPOS`
: if the pattern contains `\G`;

`plus`
if the pattern starts with a repeated char (as in `x+y`);

`implicit`
if the pattern starts with `.*`;

`with eval`
if the pattern contain eval-groups (see the section on *(?{ code })* in the *perlre* manpage);

`anchored(TYPE)`
if the pattern may match only at a handful of places (with `TYPE` being `BOL`, `MBOL`, or `GPOS`, see the table below).

If a substring is known to match at end-of-line only, it may be followed by `$`, as in `floating 'k'$`.

The optimizer-specific info is used to avoid entering (a slow) RE engine on strings which will definitely not match. If `isall` flag is set, a call to the RE engine may be avoided even when optimizer found an appropriate place for the match.

The rest of the output contains the list of *nodes* of the compiled form of the RE. Each line has format

*id*: *TYPE OPTIONAL−INFO* (*next-id*)

### Types of nodes

Here is the list of possible types with short descriptions:

```
# TYPE arg-description [num-args] [longjump-len] DESCRIPTION

# Exit points
END         no      End of program.
SUCCEED     no      Return from a subroutine, basically.

# Anchors:
BOL         no      Match "" at beginning of line.
MBOL        no      Same, assuming multiline.
SBOL        no      Same, assuming singleline.
EOS         no      Match "" at end of string.
EOL         no      Match "" at end of line.
MEOL        no      Same, assuming multiline.
SEOL        no      Same, assuming singleline.
BOUND       no      Match "" at any word boundary
BOUNDL      no      Match "" at any word boundary
NBOUND      no      Match "" at any word non-boundary
NBOUNDL     no      Match "" at any word non-boundary
GPOS        no      Matches where last m//g left off.
```

```
# [Special] alternatives
ANY         no      Match any one character (except newline).
SANY        no      Match any one character.
ANYOF       sv      Match character in (or not in) this class.
ALNUM       no      Match any alphanumeric character
ALNUML      no      Match any alphanumeric char in locale
NALNUM      no      Match any non-alphanumeric character
NALNUML     no      Match any non-alphanumeric char in locale
SPACE       no      Match any whitespace character
SPACEL      no      Match any whitespace char in locale
NSPACE      no      Match any non-whitespace character
NSPACEL     no      Match any non-whitespace char in locale
DIGIT       no      Match any numeric character
NDIGIT      no      Match any non-numeric character

# BRANCH    The set of branches constituting a single choice are hooked
#           together with their "next" pointers, since precedence prevents
#           anything being concatenated to any individual branch.  The
#           "next" pointer of the last BRANCH in a choice points to the
#           thing following the whole choice.  This is also where the
#           final "next" pointer of each individual branch points; each
#           branch starts with the operand node of a BRANCH node.
#
BRANCH      node    Match this alternative, or the next...

# BACK      Normal "next" pointers all implicitly point forward; BACK
#           exists to make loop structures possible.
# not used
BACK        no      Match "", "next" ptr points backward.

# Literals
EXACT       sv      Match this string (preceded by length).
EXACTF      sv      Match this string, folded (prec. by length).
EXACTFL     sv      Match this string, folded in locale (w/len).

# Do nothing
NOTHING     no      Match empty string.
# A variant of above which delimits a group, thus stops optimizations
TAIL        no      Match empty string. Can jump here from outside.

# STAR,PLUS '?', and complex '*' and '+', are implemented as circular
#           BRANCH structures using BACK.  Simple cases (one character
#           per match) are implemented with STAR and PLUS for speed
#           and to minimize recursive plunges.
#
STAR        node    Match this (simple) thing 0 or more times.
PLUS        node    Match this (simple) thing 1 or more times.

CURLY       sv 2    Match this simple thing {n,m} times.
CURLYN      no 2    Match next-after-this simple thing
#                   {n,m} times, set parenths.
CURLYM      no 2    Match this medium-complex thing {n,m} times.
CURLYX      sv 2    Match this complex thing {n,m} times.
```

```
# This terminator creates a loop structure for CURLYX
WHILEM      no      Do curly processing and see if rest matches.

# OPEN,CLOSE,GROUPP ...are numbered at compile time.
OPEN        num 1   Mark this point in input as start of #n.
CLOSE       num 1   Analogous to OPEN.

REF         num 1   Match some already matched string
REFF        num 1   Match already matched string, folded
REFFL       num 1   Match already matched string, folded in loc.

# grouping assertions
IFMATCH     off 1 2 Succeeds if the following matches.
UNLESSM     off 1 2 Fails if the following matches.
SUSPEND     off 1 1 "Independent" sub-RE.
IFTHEN      off 1 1 Switch, should be preceeded by switcher .
GROUPP      num 1   Whether the group matched.

# Support for long RE
LONGJMP     off 1 1 Jump far away.
BRANCHJ     off 1 1 BRANCH with long offset.

# The heavy worker
EVAL        evl 1   Execute some Perl code.

# Modifiers
MINMOD      no      Next operator is not greedy.
LOGICAL     no      Next opcode should set the flag only.

# This is not used yet
RENUM       off 1 1 Group with independently numbered parens.

# This is not really a node, but an optimized away piece of a "long" node.
# To simplify debugging output, we mark it as if it were a node
OPTIMIZED   off     Placeholder for dump.
```

### Run-time output

First of all, when doing a match, one may get no run-time output even if debugging is enabled. this means that the RE engine was never entered, all of the job was done by the optimizer.

If RE engine was entered, the output may look like this:

```
Matching '[bc]d(ef*g)+h[ij]k$' against 'abcdefg__gh__'
  Setting an EVAL scope, savestack=3
   2 <ab> <cdefg__gh_>      |  1: ANYOF
   3 <abc> <defg__gh_>      | 11: EXACT <d>
   4 <abcd> <efg__gh_>      | 13: CURLYX {1,32767}
   4 <abcd> <efg__gh_>      | 26:   WHILEM
                                0 out of 1..32767  cc=effff31c
   4 <abcd> <efg__gh_>      | 15:     OPEN1
   4 <abcd> <efg__gh_>      | 17:     EXACT <e>
   5 <abcde> <fg__gh_>      | 19:     STAR
                             EXACT <f> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
   6 <bcdef> <g__gh__>      | 22:       EXACT <g>
   7 <bcdefg> <__gh__>      | 24:       CLOSE1
   7 <bcdefg> <__gh__>      | 26:       WHILEM
                                    1 out of 1..32767  cc=effff31c
  Setting an EVAL scope, savestack=12
   7 <bcdefg> <__gh__>      | 15:         OPEN1
   7 <bcdefg> <__gh__>      | 17:         EXACT <e>
     restoring \1 to 4(4)..7
                                    failed, try continuation...
   7 <bcdefg> <__gh__>      | 27:         NOTHING
   7 <bcdefg> <__gh__>      | 28:         EXACT <h>
                                    failed...
                                failed...
```

The most significant information in the output is about the particular *node* of the compiled RE which is currently being tested against the target string. The format of these lines is

*STRING−OFFSET* <*PRE−STRING*> <*POST−STRING*> |*ID*: *TYPE*

The *TYPE* info is indented with respect to the backtracking level. Other incidental information appears interspersed within.

## NAME

perldiag − various Perl diagnostics

## DESCRIPTION

These messages are classified as follows (listed in increasing order of desperation):

```
(W) A warning (optional).
(D) A deprecation (optional).
(S) A severe warning (mandatory).
(F) A fatal error (trappable).
(P) An internal error you should never see (trappable).
(X) A very fatal error (nontrappable).
(A) An alien error message (not generated by Perl).
```

Optional warnings are enabled by using the **−w** switch. Warnings may be captured by setting `$SIG{__WARN__}` to a reference to a routine that will be called on each warning instead of printing it. See the *perlvar* manpage. Trappable errors may be trapped using the eval operator. See the `eval` entry in the *perlfunc* manpage.

Some of these messages are generic. Spots that vary are denoted with a `%s`, just as in a printf format. Note that some messages start with a `%s`! The symbols `"%(−?@` sort before the letters, while `[` and `\` sort after.

"my" variable %s can't be in a package

(F) Lexically scoped variables aren't in a package, so it doesn't make sense to try to declare one with a package qualifier on the front. Use *local()* if you want to localize a package variable.

"my" variable %s masks earlier declaration in same scope

(W) A lexical variable has been redeclared in the same scope, effectively eliminating all access to the previous instance. This is almost always a typographical error. Note that the earlier variable will still exist until the end of the scope or until all closure referents to it are destroyed.

"no" not allowed in expression

(F) The "no" keyword is recognized and executed at compile time, and returns no useful value. See the *perlmod* manpage.

"use" not allowed in expression

(F) The "use" keyword is recognized and executed at compile time, and returns no useful value. See the *perlmod* manpage.

% may only be used in unpack

(F) You can't pack a string by supplying a checksum, because the checksumming process loses information, and you can't go the other way. See the `unpack` entry in the *perlfunc* manpage.

%s (...) interpreted as function

(W) You've run afoul of the rule that says that any list operator followed by parentheses turns into a function, with all the list operators arguments found inside the parentheses. See the section on *Terms and List Operators (Leftward)* in the *perlop* manpage.

%s argument is not a HASH element

(F) The argument to *exists()* must be a hash element, such as

```
$foo{$bar}
$ref->[12]->{"susie"}
```

%s argument is not a HASH element or slice

(F) The argument to *delete()* must be either a hash element, such as

```
$foo{$bar}
$ref->[12]->{"susie"}
```

or a hash slice, such as

```
@foo{$bar, $baz, $xyzzy}
@{$ref->[12]}{"susie", "queue"}
```

%s did not return a true value

(F) A required (or used) file must return a true value to indicate that it compiled correctly and ran its initialization code correctly. It's traditional to end such a file with a "1;", though any true value would do. See the `require` entry in the *perlfunc* manpage.

%s found where operator expected

(S) The Perl lexer knows whether to expect a term or an operator. If it sees what it knows to be a term when it was expecting to see an operator, it gives you this warning. Usually it indicates that an operator or delimiter was omitted, such as a semicolon.

%s had compilation errors

(F) The final summary message when a `perl -c` fails.

%s has too many errors

(F) The parser has given up trying to parse the program after 10 errors. Further error messages would likely be uninformative.

%s matches null string many times

(W) The pattern you've specified would be an infinite loop if the regular expression engine didn't specifically check for that. See the *perlre* manpage.

%s never introduced

(S) The symbol in question was declared but somehow went out of scope before it could possibly have been used.

%s syntax OK

(F) The final summary message when a `perl -c` succeeds.

%s: Command not found

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

%s: Expression syntax

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

%s: Undefined variable

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

%s: not found

(A) You've accidentally run your script through the Bourne shell instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

(Missing semicolon on previous line?)

(S) This is an educated guess made in conjunction with the message "%s found where operator expected". Don't automatically put a semicolon on the previous line just because you saw this message.

**–P** not allowed for setuid/setgid script

(F) The script would have to be opened by the C preprocessor by name, which provides a race condition that breaks security.

`-T` and `-B` not implemented on filehandles

(F) Perl can't peek at the stdio buffer of filehandles when it doesn't know about your kind of stdio. You'll have to use a filename instead.

−p destination: %s
: (F) An error occurred during the implicit output invoked by the `−p` command-line switch. (This output goes to STDOUT unless you've redirected it with *select()*.)

500 Server error
: See Server error.

?+* follows nothing in regexp
: (F) You started a regular expression with a quantifier. Backslash it if you meant it literally. See the *perlre* manpage.

@ outside of string
: (F) You had a pack template that specified an absolute position outside the string being unpacked. See the `pack` entry in the *perlfunc* manpage.

accept() on closed fd
: (W) You tried to do an accept on a closed socket. Did you forget to check the return value of your *socket()* call? See the `accept` entry in the *perlfunc* manpage.

Allocation too large: %lx
: (X) You can't allocate more than 64K on an MS−DOS machine.

Applying %s to %s will act on scalar(%s)
: (W) The pattern match (//), substitution (s///), and transliteration (tr///) operators work on scalar values. If you apply one of them to an array or a hash, it will convert the array or hash to a scalar value -- the length of an array, or the population info of a hash -- and then work on that scalar value. This is probably not what you meant to do. See the `grep` entry in the *perlfunc* manpage and the `map` entry in the *perlfunc* manpage for alternatives.

Arg too short for msgsnd
: (F) *msgsnd()* requires a string at least as long as *sizeof*(long).

Ambiguous use of %s resolved as %s
: (W)(S) You said something that may not be interpreted the way you thought. Normally it's pretty easy to disambiguate it by supplying a missing quote, operator, parenthesis pair or declaration.

Ambiguous call resolved as CORE::%s(), qualify as such or use &
: (W) A subroutine you have declared has the same name as a Perl keyword, and you have used the name without qualification for calling one or the other. Perl decided to call the builtin because the subroutine is not imported.

  To force interpretation as a subroutine call, either put an ampersand before the subroutine name, or qualify the name with its package. Alternatively, you can import the subroutine (or pretend that it's imported with the `use subs` pragma).

  To silently interpret it as the Perl operator, use the `CORE::` prefix on the operator (e.g. `CORE::log($x)`) or by declaring the subroutine to be an object method (see the *attrs* manpage).

Args must match #! line
: (F) The setuid emulator requires that the arguments Perl was invoked with match the arguments specified on the #! line. Since some systems impose a one-argument limit on the #! line, try combining switches; for example, turn `−w −U` into `−wU`.

Argument "%s" isn't numeric%s
: (W) The indicated string was fed as an argument to an operator that expected a numeric value instead. If you're fortunate the message will identify which operator was so unfortunate.

Array @%s missing the @ in argument %d of %s()
: (D) Really old Perl let you omit the @ on array names in some spots. This is now heavily deprecated.

assertion botched: %s
: (P) The malloc package that comes with Perl had an internal failure.

Assertion failed: file "%s"

(P) A general assertion failed. The file in question must be examined.

Assignment to both a list and a scalar

(F) If you assign to a conditional operator, the 2nd and 3rd arguments must either both be scalars or both be lists. Otherwise Perl won't know which context to supply to the right side.

Attempt to free non-arena SV: 0x%lx

(P) All SV objects are supposed to be allocated from arenas that will be garbage collected on exit. An SV was discovered to be outside any of those arenas.

Attempt to free nonexistent shared string

(P) Perl maintains a reference counted internal table of strings to optimize the storage and access of hash keys and other strings. This indicates someone tried to decrement the reference count of a string that can no longer be found in the table.

Attempt to free temp prematurely

(W) Mortalized values are supposed to be freed by the *free_tmps()* routine. This indicates that something else is freeing the SV before the *free_tmps()* routine gets a chance, which means that the *free_tmps()* routine will be freeing an unreferenced scalar when it does try to free it.

Attempt to free unreferenced glob pointers

(P) The reference counts got screwed up on symbol aliases.

Attempt to free unreferenced scalar

(W) Perl went to decrement the reference count of a scalar to see if it would go to 0, and discovered that it had already gone to 0 earlier, and should have been freed, and in fact, probably was freed. This could indicate that *SvREFCNT_dec()* was called too many times, or that *SvREFCNT_inc()* was called too few times, or that the SV was mortalized when it shouldn't have been, or that memory has been corrupted.

Attempt to pack pointer to temporary value

(W) You tried to pass a temporary value (like the result of a function, or a computed expression) to the "p" *pack()* template. This means the result contains a pointer to a location that could become invalid anytime, even before the end of the current statement. Use literals or global values as arguments to the "p" *pack()* template to avoid this warning.

Attempt to use reference as lvalue in substr

(W) You supplied a reference as the first argument to *substr()* used as an lvalue, which is pretty strange. Perhaps you forgot to dereference it first. See the `substr` entry in the *perlfunc* manpage.

Bad arg length for %s, is %d, should be %d

(F) You passed a buffer of the wrong size to one of *msgctl()*, *semctl()* or *shmctl()*. In C parlance, the correct sizes are, respectively, sizeof(struct msqid_ds *), sizeof(struct semid_ds *), and sizeof(struct shmid_ds *).

Bad filehandle: %s

(F) A symbol was passed to something wanting a filehandle, but the symbol has no filehandle associated with it. Perhaps you didn't do an *open()*, or did it in another package.

Bad free() ignored

(S) An internal routine called *free()* on something that had never been *malloc()*ed in the first place. Mandatory, but can be disabled by setting environment variable `PERL_BADFREE` to 1.

This message can be quite often seen with DB_File on systems with "hard" dynamic linking, like `AIX` and `OS/2`. It is a bug of `Berkeley DB` which is left unnoticed if `DB` uses *forgiving* system *malloc()*.

Bad hash

(P) One of the internal hash routines was passed a null HV pointer.

Bad index while coercing array into hash

(F) The index looked up in the hash found as the 0'th element of a pseudo-hash is not legal. Index values must be at 1 or greater. See the *perlref* manpage.

Bad name after %s::

(F) You started to name a symbol by using a package prefix, and then didn't finish the symbol. In particular, you can't interpolate outside of quotes, so

```
$var = 'myvar';
$sym = mypack::$var;
```

is not the same as

```
$var = 'myvar';
$sym = "mypack::$var";
```

Bad symbol for array

(P) An internal request asked to add an array entry to something that wasn't a symbol table entry.

Bad symbol for filehandle

(P) An internal request asked to add a filehandle entry to something that wasn't a symbol table entry.

Bad symbol for hash

(P) An internal request asked to add a hash entry to something that wasn't a symbol table entry.

Badly placed ()'s

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

Bareword "%s" not allowed while "strict subs" in use

(F) With "strict subs" in use, a bareword is only allowed as a subroutine identifier, in curly braces or to the left of the "=>" symbol. Perhaps you need to predeclare a subroutine?

Bareword "%s" refers to nonexistent package

(W) You used a qualified bareword of the form `Foo::`, but the compiler saw no other uses of that namespace before that point. Perhaps you need to predeclare a package?

BEGIN failed—compilation aborted

(F) An untrapped exception was raised while executing a BEGIN subroutine. Compilation stops immediately and the interpreter is exited.

BEGIN not safe after errors—compilation aborted

(F) Perl found a `BEGIN {}` subroutine (or a `use` directive, which implies a `BEGIN {}`) after one or more compilation errors had already occurred. Since the intended environment for the `BEGIN {}` could not be guaranteed (due to the errors), and since subsequent code likely depends on its correct operation, Perl just gave up.

bind() on closed fd

(W) You tried to do a bind on a closed socket. Did you forget to check the return value of your *socket()* call? See the `bind` entry in the *perlfunc* manpage.

Bizarre copy of %s in %s

(P) Perl detected an attempt to copy an internal value that is not copiable.

Callback called exit

(F) A subroutine invoked from an external package via *perl_call_sv()* exited by calling exit.

Can't "goto" outside a block

(F) A "goto" statement was executed to jump out of what might look like a block, except that it isn't a proper block. This usually occurs if you tried to jump out of a *sort()* block or subroutine, which is a no-no. See the `goto` entry in the *perlfunc* manpage.

Can't "goto" into the middle of a foreach loop

(F) A "goto" statement was executed to jump into the middle of a foreach loop. You can't get there from here. See the `goto` entry in the *perlfunc* manpage.

Can't "last" outside a block

(F) A "last" statement was executed to break out of the current block, except that there's this itty bitty problem called there isn't a current block. Note that an "if" or "else" block doesn't count as a "loopish" block, as doesn't a block given to *sort()*. You can usually double the curlies to get the same effect though, because the inner curlies will be considered a block that loops once. See the `last` entry in the *perlfunc* manpage.

Can't "next" outside a block

(F) A "next" statement was executed to reiterate the current block, but there isn't a current block. Note that an "if" or "else" block doesn't count as a "loopish" block, as doesn't a block given to *sort()*. You can usually double the curlies to get the same effect though, because the inner curlies will be considered a block that loops once. See the `next` entry in the *perlfunc* manpage.

Can't "redo" outside a block

(F) A "redo" statement was executed to restart the current block, but there isn't a current block. Note that an "if" or "else" block doesn't count as a "loopish" block, as doesn't a block given to *sort()*. You can usually double the curlies to get the same effect though, because the inner curlies will be considered a block that loops once. See the `redo` entry in the *perlfunc* manpage.

Can't bless non-reference value

(F) Only hard references may be blessed. This is how Perl "enforces" encapsulation of objects. See the *perlobj* manpage.

Can't break at that line

(S) A warning intended to only be printed while running within the debugger, indicating the line number specified wasn't the location of a statement that could be stopped at.

Can't call method "%s" in empty package "%s"

(F) You called a method correctly, and it correctly indicated a package functioning as a class, but that package doesn't have ANYTHING defined in it, let alone methods. See the *perlobj* manpage.

Can't call method "%s" on unblessed reference

(F) A method call must know in what package it's supposed to run. It ordinarily finds this out from the object reference you supply, but you didn't supply an object reference in this case. A reference isn't an object reference until it has been blessed. See the *perlobj* manpage.

Can't call method "%s" without a package or object reference

(F) You used the syntax of a method call, but the slot filled by the object reference or package name contains an expression that returns a defined value which is neither an object reference nor a package name. Something like this will reproduce the error:

```
$BADREF = 42;
process $BADREF 1,2,3;
$BADREF->process(1,2,3);
```

Can't call method "%s" on an undefined value

(F) You used the syntax of a method call, but the slot filled by the object reference or package name contains an undefined value. Something like this will reproduce the error:

```
$BADREF = undef;
process $BADREF 1,2,3;
$BADREF->process(1,2,3);
```

Can't chdir to %s

(F) You called `perl -x/foo/bar`, but `/foo/bar` is not a directory that you can chdir to, possibly because it doesn't exist.

Can't coerce %s to integer in %s

(F) Certain types of SVs, in particular real symbol table entries (typeglobs), can't be forced to stop being what they are. So you can't say things like:

```
*foo += 1;
```

You CAN say

```
$foo = *foo;
$foo += 1;
```

but then `$foo` no longer contains a glob.

Can't coerce %s to number in %s

(F) Certain types of SVs, in particular real symbol table entries (typeglobs), can't be forced to stop being what they are.

Can't coerce %s to string in %s

(F) Certain types of SVs, in particular real symbol table entries (typeglobs), can't be forced to stop being what they are.

Can't coerce array into hash

(F) You used an array where a hash was expected, but the array has no information on how to map from keys to array indices. You can do that only with arrays that have a hash reference at index 0.

Can't create pipe mailbox

(P) An error peculiar to VMS. The process is suffering from exhausted quotas or other plumbing problems.

Can't declare %s in my

(F) Only scalar, array, and hash variables may be declared as lexical variables. They must have ordinary identifiers as names.

Can't do inplace edit on %s: %s

(S) The creation of the new file failed for the indicated reason.

Can't do inplace edit without backup

(F) You're on a system such as MS−DOS that gets confused if you try reading from a deleted (but still opened) file. You have to say `-i.bak`, or some such.

Can't do inplace edit: %s > 14 characters

(S) There isn't enough room in the filename to make a backup name for the file.

Can't do inplace edit: %s is not a regular file

(S) You tried to use the **−i** switch on a special file, such as a file in /dev, or a FIFO. The file was ignored.

Can't do setegid!

(P) The *setegid()* call failed for some reason in the setuid emulator of suidperl.

Can't do seteuid!

(P) The setuid emulator of suidperl failed for some reason.

Can't do setuid

(F) This typically means that ordinary perl tried to exec suidperl to do setuid emulation, but couldn't exec it. It looks for a name of the form sperl5.000 in the same directory that the perl executable resides under the name perl5.000, typically /usr/local/bin on Unix machines. If the file is there, check the execute permissions. If it isn't, ask your sysadmin why he and/or she removed it.

Can't do waitpid with flags

(F) This machine doesn't have either *waitpid()* or *wait4()*, so only *waitpid()* without flags is emulated.

Can't do {n,m} with n > m

(F) Minima must be less than or equal to maxima. If you really want your regexp to match something 0 times, just put {0}. See the *perlre* manpage.

Can't emulate –%s on #! line

(F) The #! line specifies a switch that doesn't make sense at this point. For example, it'd be kind of silly to put a **–x** on the #! line.

Can't exec "%s": %s

(W) An *system()*, *exec()*, or piped open call could not execute the named program for the indicated reason. Typical reasons include: the permissions were wrong on the file, the file wasn't found in `$ENV{PATH}`, the executable in question was compiled for another architecture, or the #! line in a script points to an interpreter that can't be run for similar reasons. (Or maybe your system doesn't support #! at all.)

Can't exec %s

(F) Perl was trying to execute the indicated program for you because that's what the #! line said. If that's not what you wanted, you may need to mention "perl" on the #! line somewhere.

Can't execute %s

(F) You used the **–S** switch, but the copies of the script to execute found in the PATH did not have correct permissions.

Can't find %s on PATH, '.' not in PATH

(F) You used the **–S** switch, but the script to execute could not be found in the PATH, or at least not with the correct permissions. The script exists in the current directory, but PATH prohibits running it.

Can't find %s on PATH

(F) You used the **–S** switch, but the script to execute could not be found in the PATH.

Can't find label %s

(F) You said to goto a label that isn't mentioned anywhere that it's possible for us to go to. See the `goto` entry in the *perlfunc* manpage.

Can't find string terminator %s anywhere before EOF

(F) Perl strings can stretch over multiple lines. This message means that the closing delimiter was omitted. Because bracketed quotes count nesting levels, the following is missing its final parenthesis:

```
print q(The character '(' starts a side comment.);
```

If you're getting this error from a here-document, you may have included unseen whitespace before or after your closing tag. A good programmer's editor will have a way to help you find these characters.

Can't fork

(F) A fatal error occurred while trying to fork while opening a pipeline.

Can't get filespec – stale stat buffer?

(S) A warning peculiar to VMS. This arises because of the difference between access checks under VMS and under the Unix model Perl assumes. Under VMS, access checks are done by filename, rather than by bits in the stat buffer, so that ACLs and other protections can be taken into account. Unfortunately, Perl assumes that the stat buffer contains all the necessary information, and passes it, instead of the filespec, to the access checking routine. It will try to retrieve the filespec using the device name and FID present in the stat buffer, but this works only if you haven't made a subsequent call to the CRTL *stat()* routine, because the device name is overwritten with each call. If this warning appears, the name lookup failed, and the access checking routine gave up and returned FALSE, just to be conservative. (Note: The access checking routine knows about the Perl `stat` operator and file tests, so you shouldn't ever see this warning in response to a Perl command; it arises only if some internal code takes stat buffers lightly.)

Can't get pipe mailbox device name

(P) An error peculiar to VMS. After creating a mailbox to act as a pipe, Perl can't retrieve its name for later use.

Can't get SYSGEN parameter value for MAXBUF

(P) An error peculiar to VMS. Perl asked `$GETSYI` how big you want your mailbox buffers to be, and didn't get an answer.

Can't goto subroutine outside a subroutine

(F) The deeply magical "goto subroutine" call can only replace one subroutine call for another. It can't manufacture one out of whole cloth. In general you should be calling it out of only an AUTOLOAD routine anyway. See the `goto` entry in the *perlfunc* manpage.

Can't goto subroutine from an eval-string

(F) The "goto subroutine" call can't be used to jump out of an eval "string". (You can use it to jump out of an eval {BLOCK}, but you probably don't want to.)

Can't localize through a reference

(F) You said something like `local $$ref`, which Perl can't currently handle, because when it goes to restore the old value of whatever `$ref` pointed to after the scope of the *local()* is finished, it can't be sure that `$ref` will still be a reference.

Can't localize lexical variable %s

(F) You used local on a variable name that was previously declared as a lexical variable using "my". This is not allowed. If you want to localize a package variable of the same name, qualify it with the package name.

Can't localize pseudo-hash element

(F) You said something like `local $ar->{'key'}`, where `$ar` is a reference to a pseudo-hash. That hasn't been implemented yet, but you can get a similar effect by localizing the corresponding array element directly -- `local $ar->[$ar->[0]{'key'}]`.

Can't locate auto/%s.al in @INC

(F) A function (or method) was called in a package which allows autoload, but there is no function to autoload. Most probable causes are a misprint in a function/method name or a failure to `AutoSplit` the file, say, by doing `make install`.

Can't locate %s in @INC

(F) You said to do (or require, or use) a file that couldn't be found in any of the libraries mentioned in `@INC`. Perhaps you need to set the PERL5LIB or PERL5OPT environment variable to say where the extra library is, or maybe the script needs to add the library name to `@INC`. Or maybe you just misspelled the name of the file. See the `require` entry in the *perlfunc* manpage.

Can't locate object method "%s" via package "%s"

(F) You called a method correctly, and it correctly indicated a package functioning as a class, but that package doesn't define that particular method, nor does any of its base classes. See the *perlobj* manpage.

Can't locate package %s for @%s::ISA

(W) The `@ISA` array contained the name of another package that doesn't seem to exist.

Can't make list assignment to \%ENV on this system

(F) List assignment to `%ENV` is not supported on some systems, notably VMS.

Can't modify %s in %s

(F) You aren't allowed to assign to the item indicated, or otherwise try to change it, such as with an auto-increment.

Can't modify nonexistent substring

(P) The internal routine that does assignment to a *substr()* was handed a NULL.

Can't msgrcv to read-only var

(F) The target of a msgrcv must be modifiable to be used as a receive buffer.

Can't open %s: %s

(S) The implicit opening of a file through use of the <> filehandle, either implicitly under the `-n` or `-p` command-line switches, or explicitly, failed for the indicated reason. Usually this is because you don't have read permission for a file which you named on the command line.

Can't open bidirectional pipe

(W) You tried to say `open(CMD, "|cmd|")`, which is not supported. You can try any of several modules in the Perl library to do this, such as IPC::Open2. Alternately, direct the pipe's output to a file using ">", and then read it in under a different file handle.

Can't open error file %s as stderr

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the file specified after '2>' or '2>>' on the command line for writing.

Can't open input file %s as stdin

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the file specified after '<' on the command line for reading.

Can't open output file %s as stdout

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the file specified after '>' or '>>' on the command line for writing.

Can't open output pipe (name: %s)

(P) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the pipe into which to send data destined for stdout.

Can't open perl script "%s": %s

(F) The script you specified can't be opened for the indicated reason.

Can't redefine active sort subroutine %s

(F) Perl optimizes the internal handling of sort subroutines and keeps pointers into them. You tried to redefine one such sort subroutine when it was currently active, which is not allowed. If you really want to do this, you should write `sort { &func } @x` instead of `sort func @x`.

Can't rename %s to %s: %s, skipping file

(S) The rename done by the **-i** switch failed for some reason, probably because you don't have write permission to the directory.

Can't reopen input pipe (name: %s) in binary mode

(P) An error peculiar to VMS. Perl thought stdin was a pipe, and tried to reopen it to accept binary data. Alas, it failed.

Can't reswap uid and euid

(P) The *setreuid()* call failed for some reason in the setuid emulator of suidperl.

Can't return outside a subroutine

(F) The return statement was executed in mainline code, that is, where there was no subroutine call to return out of. See the *perlsub* manpage.

Can't stat script "%s"

(P) For some reason you can't *fstat()* the script even though you have it open already. Bizarre.

Can't swap uid and euid

(P) The *setreuid()* call failed for some reason in the setuid emulator of suidperl.

Can't take log of %g

(F) For ordinary real numbers, you can't take the logarithm of a negative number or zero. There's a Math::Complex package that comes standard with Perl, though, if you really want to do that for the negative numbers.

Can't take sqrt of %g

(F) For ordinary real numbers, you can't take the square root of a negative number. There's a Math::Complex package that comes standard with Perl, though, if you really want to do that.

Can't undef active subroutine

(F) You can't undefine a routine that's currently running. You can, however, redefine it while it's running, and you can even undef the redefined subroutine while the old routine is running. Go figure.

Can't unshift

(F) You tried to unshift an "unreal" array that can't be unshifted, such as the main Perl stack.

Can't upgrade that kind of scalar

(P) The internal sv_upgrade routine adds "members" to an SV, making it into a more specialized kind of SV. The top several SV types are so specialized, however, that they cannot be interconverted. This message indicates that such a conversion was attempted.

Can't upgrade to undef

(P) The undefined SV is the bottom of the totem pole, in the scheme of upgradability. Upgrading to undef indicates an error in the code calling sv_upgrade.

Can't use %%! because Errno.pm is not available

(F) The first time the %! hash is used, perl automatically loads the Errno.pm module. The Errno module is expected to tie the %! hash to provide symbolic names for `$!` errno values.

Can't use "my %s" in sort comparison

(F) The global variables `$a` and `$b` are reserved for sort comparisons. You mentioned `$a` or `$b` in the same line as the <=> or cmp operator, and the variable had earlier been declared as a lexical variable. Either qualify the sort variable with the package name, or rename the lexical variable.

Can't use %s for loop variable

(F) Only a simple scalar variable may be used as a loop variable on a foreach.

Can't use %s ref as %s ref

(F) You've mixed up your reference types. You have to dereference a reference of the type needed. You can use the *ref()* function to test the type of the reference, if need be.

Can't use \1 to mean $1 in expression

(W) In an ordinary expression, backslash is a unary operator that creates a reference to its argument. The use of backslash to indicate a backreference to a matched substring is valid only as part of a regular expression pattern. Trying to do this in ordinary Perl code produces a value that prints out looking like *SCALAR* (0xdecaf). Use the `$1` form instead.

Can't use bareword '%s' as %s ref while \"strict refs\" in use

(F) Only hard references are allowed by "strict refs". Symbolic references are disallowed. See the *perlref* manpage.

Can't use string '%s' as %s ref while "strict refs" in use

(F) Only hard references are allowed by "strict refs". Symbolic references are disallowed. See the *perlref* manpage.

Can't use an undefined value as %s reference

(F) A value used as either a hard reference or a symbolic reference must be a defined value. This helps to delurk some insidious errors.

Can't use global %s in "my"

(F) You tried to declare a magical variable as a lexical variable. This is not allowed, because the magic can be tied to only one location (namely the global variable) and it would be incredibly confusing to have variables in your program that looked like magical variables but weren't.

Can't use subscript on %s

(F) The compiler tried to interpret a bracketed expression as a subscript. But to the left of the brackets was an expression that didn't look like an array reference, or anything else subscriptable.

Can't x= to read-only value

(F) You tried to repeat a constant value (often the undefined value) with an assignment operator, which implies modifying the value itself. Perhaps you need to copy the value to a temporary, and repeat that.

Cannot find an opnumber for "%s"

(F) A string of a form `CORE::word` was given to *prototype()*, but there is no builtin with the name `word`.

Cannot resolve method '%s' overloading '%s' in package '%s'

(F|P) Error resolving overloading specified by a method name (as opposed to a subroutine reference): no such method callable via the package. If method name is `???`, this is an internal error.

Character class syntax [. .] is reserved for future extensions

(W) Within regular expression character classes ([]) the syntax beginning with "[." and ending with ".]" is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: "\[." and ".\]".

Character class syntax [: :] is reserved for future extensions

(W) Within regular expression character classes ([]) the syntax beginning with "[:" and ending with ":]" is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: "\[:" and ":\]".

Character class syntax [= =] is reserved for future extensions

(W) Within regular expression character classes ([]) the syntax beginning with "[=" and ending with "=]" is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: "\[=" and "=\]".

chmod: mode argument is missing initial 0

(W) A novice will sometimes say

```
chmod 777, $filename
```

not realizing that 777 will be interpreted as a decimal number, equivalent to 01411. Octal constants are introduced with a leading 0 in Perl, as in C.

Close on unopened file <%s>

(W) You tried to close a filehandle that was never opened.

Compilation failed in require

(F) Perl could not compile a file specified in a `require` statement. Perl uses this generic message when none of the errors that it encountered were severe enough to halt compilation immediately.

Complex regular subexpression recursion limit (%d) exceeded

(W) The regular expression engine uses recursion in complex situations where back-tracking is required. Recursion depth is limited to 32766, or perhaps less in architectures where the stack cannot grow arbitrarily. ("Simple" and "medium" situations are handled without recursion and are not subject to a limit.) Try shortening the string under examination; looping in Perl code (e.g. with `while`) rather than in the regular expression engine; or rewriting the regular expression so that it is simpler or backtracks less. (See the *perlbook* manpage for information on *Mastering Regular Expressions*.)

connect() on closed fd

(W) You tried to do a connect on a closed socket. Did you forget to check the return value of your *socket()* call? See the `connect` entry in the *perlfunc* manpage.

Constant subroutine %s redefined

(S) You redefined a subroutine which had previously been eligible for inlining. See the section on *Constant Functions* in the *perlsub* manpage for commentary and workarounds.

Constant subroutine %s undefined

(S) You undefined a subroutine which had previously been eligible for inlining. See the section on *Constant Functions* in the *perlsub* manpage for commentary and workarounds.

Copy method did not return a reference

(F) The method which overloads "=" is buggy. See the section on *Copy Constructor* in the *overload* manpage.

Corrupt malloc ptr 0x%lx at 0x%lx

(P) The malloc package that comes with Perl had an internal failure.

corrupted regexp pointers

(P) The regular expression engine got confused by what the regular expression compiler gave it.

corrupted regexp program

(P) The regular expression engine got passed a regexp program without a valid magic number.

Deep recursion on subroutine "%s"

(W) This subroutine has called itself (directly or indirectly) 100 times more than it has returned. This probably indicates an infinite recursion, unless you're writing strange benchmark programs, in which case it indicates something else.

Delimiter for here document is too long

(F) In a here document construct like `<<FOO`, the label `FOO` is too long for Perl to handle. You have to be seriously twisted to write code that triggers this error.

Did you mean &%s instead?

(W) You probably referred to an imported subroutine &FOO as `$FOO` or some such.

Did you mean $ or @ instead of %?

(W) You probably said `%hash{$key}` when you meant `$hash{$key}` or `@hash{@keys}`. On the other hand, maybe you just meant `%hash` and got carried away.

Died

(F) You passed *die()* an empty string (the equivalent of `die ""`) or you called it with no args and both `$@` and `$_` were empty.

Do you need to predeclare %s?

(S) This is an educated guess made in conjunction with the message "%s found where operator expected". It often means a subroutine or module name is being referenced that hasn't been declared yet. This may be because of ordering problems in your file, or because of a missing "sub", "package", "require", or "use" statement. If you're referencing something that isn't defined yet, you don't actually have to define the subroutine or package before the current location. You can use an empty "sub foo;" or "package FOO;" to enter a "forward" declaration.

Don't know how to handle magic of type '%s'

(P) The internal handling of magical variables has been cursed.

do_study: out of memory

(P) This should have been caught by *safemalloc()* instead.

Duplicate free() ignored

(S) An internal routine called *free()* on something that had already been freed.

elseif should be elsif

(S) There is no keyword "elseif" in Perl because Larry thinks it's ugly. Your code will be interpreted as an attempt to call a method named "elseif" for the class returned by the following block. This is unlikely to be what you want.

END failed—cleanup aborted

(F) An untrapped exception was raised while executing an END subroutine. The interpreter is immediately exited.

Error converting file specification %s

(F) An error peculiar to VMS. Because Perl may have to deal with file specifications in either VMS or Unix syntax, it converts them to a single form when it must operate on them directly. Either you've passed an invalid file specification to Perl, or you've found a case the conversion routines don't handle.

Drat.

%s: Eval-group in insecure regular expression

(F) Perl detected tainted data when trying to compile a regular expression that contains the `(?{ ... })` zero-width assertion, which is unsafe. See the section on *(?{ code })* in the *perlre* manpage, and the *perlsec* manpage.

%s: Eval-group not allowed, use re 'eval'

(F) A regular expression contained the `(?{ ... })` zero-width assertion, but that construct is only allowed when the `use re 'eval'` pragma is in effect. See the section on *(?{ code })* in the *perlre* manpage.

%s: Eval-group not allowed at run time

(F) Perl tried to compile a regular expression containing the `(?{ ... })` zero-width assertion at run time, as it would when the pattern contains interpolated values. Since that is a security risk, it is not allowed. If you insist, you may still do this by explicitly building the pattern from an interpolated string at run time and using that in an *eval()*. See the section on *(?{ code })* in the *perlre* manpage.

Excessively long <> operator

(F) The contents of a <> operator may not exceed the maximum size of a Perl identifier. If you're just trying to glob a long list of filenames, try using the *glob()* operator, or put the filenames into a variable and glob that.

Execution of %s aborted due to compilation errors

(F) The final summary message when a Perl compilation fails.

Exiting eval via %s

(W) You are exiting an eval by unconventional means, such as a goto, or a loop control statement.

Exiting pseudo-block via %s

(W) You are exiting a rather special block construct (like a sort block or subroutine) by unconventional means, such as a goto, or a loop control statement. See the `sort` entry in the *perlfunc* manpage.

Exiting subroutine via %s

(W) You are exiting a subroutine by unconventional means, such as a goto, or a loop control statement.

Exiting substitution via %s

(W) You are exiting a substitution by unconventional means, such as a return, a goto, or a loop control statement.

Explicit blessing to '' (assuming package main)

(W) You are blessing a reference to a zero length string. This has the effect of blessing the reference into the package main. This is usually not what you want. Consider providing a default target package, e.g. *bless*($ref, `$p` or 'MyPackage');

Fatal VMS error at %s, line %d

(P) An error peculiar to VMS. Something untoward happened in a VMS system service or RTL routine; Perl's exit status should provide more details. The filename in "at `%s`" and the line number in "line `%d`" tell you which section of the Perl source code is distressed.

fcntl is not implemented

(F) Your machine apparently doesn't implement *fcntl()*. What is this, a PDP−11 or something?

Filehandle %s never opened

(W) An I/O operation was attempted on a filehandle that was never initialized. You need to do an *open()* or a *socket()* call, or call a constructor from the FileHandle package.

Filehandle %s opened for only input

(W) You tried to write on a read-only filehandle. If you intended it to be a read-write filehandle, you needed to open it with "+<" or "+>" or "+>>" instead of with "<" or nothing. If you intended only to write the file, use ">" or ">>". See the `open` entry in the *perlfunc* manpage.

Filehandle opened for only input

(W) You tried to write on a read-only filehandle. If you intended it to be a read-write filehandle, you needed to open it with "+<" or "+>" or "+>>" instead of with "<" or nothing. If you intended only to write the file, use ">" or ">>". See the `open` entry in the *perlfunc* manpage.

Final $ should be \$ or $name

(F) You must now decide whether the final $ in a string was meant to be a literal dollar sign, or was meant to introduce a variable name that happens to be missing. So you have to put either the backslash or the name.

Final @ should be \@ or @name

(F) You must now decide whether the final @ in a string was meant to be a literal "at" sign, or was meant to introduce a variable name that happens to be missing. So you have to put either the backslash or the name.

Format %s redefined

(W) You redefined a format. To suppress this warning, say

```
{
    local $^W = 0;
    eval "format NAME =...";
}
```

Format not terminated

(F) A format must be terminated by a line with a solitary dot. Perl got to the end of your file without finding such a line.

Found = in conditional, should be ==

(W) You said

```
if ($foo = 123)
```

when you meant

```
if ($foo == 123)
```

(or something like that).

gdbm store returned %d, errno %d, key "%s"

(S) A warning from the GDBM_File extension that a store failed.

gethostent not implemented

(F) Your C library apparently doesn't implement *gethostent()*, probably because if it did, it'd feel morally obligated to return every hostname on the Internet.

get{sock,peer}name() on closed fd

(W) You tried to get a socket or peer socket name on a closed socket. Did you forget to check the return value of your *socket()* call?

getpwnam returned invalid UIC %#o for user "%s"

(S) A warning peculiar to VMS. The call to `sys$getuai` underlying the `getpwnam` operator returned an invalid UIC.

Glob not terminated

(F) The lexer saw a left angle bracket in a place where it was expecting a term, so it's looking for the corresponding right angle bracket, and not finding it. Chances are you left some needed parentheses out earlier in the line, and you really meant a "less than".

Global symbol "%s" requires explicit package name

(F) You've said "use strict vars", which indicates that all variables must either be lexically scoped (using "my"), or explicitly qualified to say which package the global variable is in (using "::").

goto must have label

(F) Unlike with "next" or "last", you're not allowed to goto an unspecified destination. See the `goto` entry in the *perlfunc* manpage.

Had to create %s unexpectedly

(S) A routine asked for a symbol from a symbol table that ought to have existed already, but for some reason it didn't, and had to be created on an emergency basis to prevent a core dump.

Hash %%s missing the % in argument %d of %s()

(D) Really old Perl let you omit the % on hash names in some spots. This is now heavily deprecated.

Identifier too long

(F) Perl limits identifiers (names for variables, functions, etc.) to about 250 characters for simple names, and somewhat more for compound names (like `$A::B`). You've exceeded Perl's limits. Future versions of Perl are likely to eliminate these arbitrary limitations.

Ill-formed logical name |%s| in prime_env_iter

(W) A warning peculiar to VMS. A logical name was encountered when preparing to iterate over `%ENV` which violates the syntactic rules governing logical names. Because it cannot be translated normally, it is skipped, and will not appear in `%ENV`. This may be a benign occurrence, as some software packages might directly modify logical name tables and introduce nonstandard names, or it may indicate that a logical name table has been corrupted.

Illegal character %s (carriage return)

(F) A carriage return character was found in the input. This is an error, and not a warning, because carriage return characters can break multi-line strings, including here documents (e.g., `print <<EOF;`).

Under Unix, this error is usually caused by executing Perl code -- either the main program, a module, or an eval'd string -- that was transferred over a network connection from a non-Unix system without properly converting the text file format.

Under systems that use something other than '\n' to delimit lines of text, this error can also be caused by reading Perl code from a file handle that is in binary mode (as set by the `binmode` operator).

In either case, the Perl code in question will probably need to be converted with something like `s/\x0D\x0A?/\n/g` before it can be executed.

Illegal division by zero

(F) You tried to divide a number by 0. Either something was wrong in your logic, or you need to put a conditional in to guard against meaningless input.

Illegal modulus zero

(F) You tried to divide a number by 0 to get the remainder. Most numbers don't take to this kindly.

Illegal octal digit

(F) You used an 8 or 9 in a octal number.

Illegal octal digit ignored

(W) You may have tried to use an 8 or 9 in a octal number. Interpretation of the octal number stopped before the 8 or 9.

Illegal hex digit ignored

(W) You may have tried to use a character other than 0 – 9 or A – F in a hexadecimal number. Interpretation of the hexadecimal number stopped before the illegal character.

Illegal switch in PERL5OPT: %s

(X) The PERL5OPT environment variable may only be used to set the following switches: **–[DIMUdmw]**.

In string, @%s now must be written as \@%s

(F) It used to be that Perl would try to guess whether you wanted an array interpolated or a literal @. It did this when the string was first used at runtime. Now strings are parsed at compile time, and ambiguous instances of @ must be disambiguated, either by prepending a backslash to indicate a literal, or by declaring (or using) the array within the program before the string (lexically). (Someday it will simply assume that an unbackslashed @ interpolates an array.)

Insecure dependency in %s

(F) You tried to do something that the tainting mechanism didn't like. The tainting mechanism is turned on when you're running setuid or setgid, or when you specify **−T** to turn it on explicitly. The tainting mechanism labels all data that's derived directly or indirectly from the user, who is considered to be unworthy of your trust. If any such data is used in a "dangerous" operation, you get this error. See the *perlsec* manpage for more information.

Insecure directory in %s

(F) You can't use *system()*, *exec()*, or a piped open in a setuid or setgid script if `$ENV{PATH}` contains a directory that is writable by the world. See the *perlsec* manpage.

Insecure $ENV{%s} while running %s

(F) You can't use *system()*, *exec()*, or a piped open in a setuid or setgid script if any of `$ENV{PATH}`, `$ENV{IFS}`, `$ENV{CDPATH}`, `$ENV{ENV}` or `$ENV{BASH_ENV}` are derived from data supplied (or potentially supplied) by the user. The script must set the path to a known value, using trustworthy data. See the *perlsec* manpage.

Integer overflow in hex number

(S) The literal hex number you have specified is too big for your architecture. On a 32-bit architecture the largest hex literal is 0xFFFFFFFF.

Integer overflow in octal number

(S) The literal octal number you have specified is too big for your architecture. On a 32-bit architecture the largest octal literal is 037777777777.

Internal inconsistency in tracking vforks

(S) A warning peculiar to VMS. Perl keeps track of the number of times you've called `fork` and `exec`, to determine whether the current call to `exec` should affect the current script or a subprocess (see the `exec` entry in the *perlvms* manpage). Somehow, this count has become scrambled, so Perl is making a guess and treating this `exec` as a request to terminate the Perl script and execute the specified command.

internal disaster in regexp

(P) Something went badly wrong in the regular expression parser.

internal error: glob failed

(P) Something went wrong with the external *program* (s) used for `glob` and `<*.c>`. This may mean that your csh (C shell) is broken. If so, you should change all of the csh-related variables in config.sh: If you have tcsh, make the variables refer to it as if it were csh (e.g. `full_csh='/usr/bin/tcsh'`); otherwise, make them all empty (except that `d_csh` should be `'undef'`) so that Perl will think csh is missing. In either case, after editing config.sh, run `./Configure -S` and rebuild Perl.

internal urp in regexp at /%s/

(P) Something went badly awry in the regular expression parser.

invalid [] range in regexp

(F) The range specified in a character class had a minimum character greater than the maximum character. See the *perlre* manpage.

Invalid conversion in %s: "%s"

(W) Perl does not understand the given format conversion. See the `sprintf` entry in the *perlfunc* manpage.

Invalid type in pack: '%s'

(F) The given character is not a valid pack type. See the `pack` entry in the *perlfunc* manpage. (W) The given character is not a valid pack type but used to be silently ignored.

Invalid type in unpack: '%s'

(F) The given character is not a valid unpack type. See the `unpack` entry in the *perlfunc* manpage. (W) The given character is not a valid unpack type but used to be silently ignored.

ioctl is not implemented

(F) Your machine apparently doesn't implement *ioctl()*, which is pretty strange for a machine that supports C.

junk on end of regexp

(P) The regular expression parser is confused.

Label not found for "last %s"

(F) You named a loop to break out of, but you're not currently in a loop of that name, not even if you count where you were called from. See the `last` entry in the *perlfunc* manpage.

Label not found for "next %s"

(F) You named a loop to continue, but you're not currently in a loop of that name, not even if you count where you were called from. See the `last` entry in the *perlfunc* manpage.

Label not found for "redo %s"

(F) You named a loop to restart, but you're not currently in a loop of that name, not even if you count where you were called from. See the `last` entry in the *perlfunc* manpage.

listen() on closed fd

(W) You tried to do a listen on a closed socket. Did you forget to check the return value of your *socket()* call? See the `listen` entry in the *perlfunc* manpage.

Method for operation %s not found in package %s during blessing

(F) An attempt was made to specify an entry in an overloading table that doesn't resolve to a valid subroutine. See the *overload* manpage.

Might be a runaway multi-line %s string starting on line %d

(S) An advisory indicating that the previous error may have been caused by a missing delimiter on a string or pattern, because it eventually ended earlier on the current line.

Misplaced _ in number

(W) An underline in a decimal constant wasn't on a 3-digit boundary.

Missing $ on loop variable

(F) Apparently you've been programming in **csh** too much. Variables are always mentioned with the $ in Perl, unlike in the shells, where it can vary from one line to the next.

Missing comma after first argument to %s function

(F) While certain functions allow you to specify a filehandle or an "indirect object" before the argument list, this ain't one of them.

Missing operator before %s?

(S) This is an educated guess made in conjunction with the message "%s found where operator expected". Often the missing operator is a comma.

Missing right bracket

(F) The lexer counted more opening curly brackets (braces) than closing ones. As a general rule, you'll find it's missing near the place you were last editing.

Modification of a read-only value attempted

(F) You tried, directly or indirectly, to change the value of a constant. You didn't, of course, try "2 = 1", because the compiler catches that. But an easy way to do the same thing is:

```
sub mod { $_[0] = 1 }
mod(2);
```

Another way is to assign to a *substr()* that's off the end of the string.

Modification of non-creatable array value attempted, subscript %d

(F) You tried to make an array value spring into existence, and the subscript was probably negative, even counting from end of the array backwards.

Modification of non-creatable hash value attempted, subscript "%s"

(P) You tried to make a hash value spring into existence, and it couldn't be created for some peculiar reason.

Module name must be constant

(F) Only a bare module name is allowed as the first argument to a "use".

msg%s not implemented

(F) You don't have System V message IPC on your system.

Multidimensional syntax %s not supported

(W) Multidimensional arrays aren't written like `$foo[1,2,3]`. They're written like `$foo[1][2][3]`, as in C.

Name "%s::%s" used only once: possible typo

(W) Typographical errors often show up as unique variable names. If you had a good reason for having a unique name, then just mention it again somehow to suppress the message. The `use vars` pragma is provided for just this purpose.

Negative length

(F) You tried to do a read/write/send/recv operation with a buffer length that is less than 0. This is difficult to imagine.

nested *?+ in regexp

(F) You can't quantify a quantifier without intervening parentheses. So things like ** or +* or ?* are illegal.

Note, however, that the minimal matching quantifiers, `*?`, `+?`, and `??` appear to be nested quantifiers, but aren't. See the *perlre* manpage.

No #! line

(F) The setuid emulator requires that scripts have a well-formed #! line even on machines that don't support the #! construct.

No %s allowed while running setuid

(F) Certain operations are deemed to be too insecure for a setuid or setgid script to even be allowed to attempt. Generally speaking there will be another way to do what you want that is, if not secure, at least securable. See the *perlsec* manpage.

No **−e** allowed in setuid scripts

(F) A setuid script can't be specified by the user.

No comma allowed after %s

(F) A list operator that has a filehandle or "indirect object" is not allowed to have a comma between that and the following arguments. Otherwise it'd be just another one of the arguments.

One possible cause for this is that you expected to have imported a constant to your name space with **use** or **import** while no such importing took place, it may for example be that your operating system does not support that particular constant. Hopefully you did use an explicit import list for the constants you expect to see, please see the `use` entry in the *perlfunc* manpage and the `import` entry in the *perlfunc* manpage. While an explicit import list would probably have caught this error earlier it naturally does not remedy the fact that your operating system still does not support that constant. Maybe you have a typo in the constants of the symbol import list of **use** or **import** or in the constant name at the line where this error was triggered?

No command into which to pipe on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '`|`' at the end of the command line, so it doesn't know where you want to pipe the output from this command.

No DB::DB routine defined

(F) The currently executing code was compiled with the **–d** switch, but for some reason the perl5db.pl file (or some facsimile thereof) didn't define a routine to be called at the beginning of each statement. Which is odd, because the file should have been required automatically, and should have blown up the require if it didn't parse right.

No dbm on this machine

(P) This is counted as an internal error, because every machine should supply dbm nowadays, because Perl comes with SDBM. See the *SDBM_File* manpage.

No DBsub routine

(F) The currently executing code was compiled with the **–d** switch, but for some reason the perl5db.pl file (or some facsimile thereof) didn't define a DB::sub routine to be called at the beginning of each ordinary subroutine call.

No error file after 2> or 2>> on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '2>' or a '2>>' on the command line, but can't find the name of the file to which to write data destined for stderr.

No input file after < on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '<' on the command line, but can't find the name of the file from which to read data for stdin.

No output file after > on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a lone '>' at the end of the command line, so it doesn't know where you wanted to redirect stdout.

No output file after > or >> on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '>' or a '>>' on the command line, but can't find the name of the file to which to write data destined for stdout.

No Perl script found in input

(F) You called `perl -x`, but no line was found in the file beginning with #! and containing the word "perl".

No setregid available

(F) Configure didn't find anything resembling the *setregid()* call for your system.

No setreuid available

(F) Configure didn't find anything resembling the *setreuid()* call for your system.

No space allowed after **–I**

(F) The argument to **–I** must follow the **–I** immediately with no intervening space.

No such array field

(F) You tried to access an array as a hash, but the field name used is not defined. The hash at index 0 should map all valid field names to array indices for that to work.

No such field "%s" in variable %s of type %s

(F) You tried to access a field of a typed variable where the type does not know about the field name. The field names are looked up in the `%FIELDS` hash in the type package at compile time. The `%FIELDS` hash is usually set up with the 'fields' pragma.

No such pipe open

(P) An error peculiar to VMS. The internal routine *my_pclose()* tried to close a pipe which hadn't been opened. This should have been caught earlier as an attempt to close an unopened filehandle.

No such signal: SIG%s

(W) You specified a signal name as a subscript to `%SIG` that was not recognized. Say `kill -l` in your shell to see the valid signal names on your system.

Not a CODE reference
: (F) Perl was trying to evaluate a reference to a code value (that is, a subroutine), but found a reference to something else instead. You can use the *ref()* function to find out what kind of ref it really was. See also the *perlref* manpage.

Not a format reference
: (F) I'm not sure how you managed to generate a reference to an anonymous format, but this indicates you did, and that it didn't exist.

Not a GLOB reference
: (F) Perl was trying to evaluate a reference to a "typeglob" (that is, a symbol table entry that looks like `*foo`), but found a reference to something else instead. You can use the *ref()* function to find out what kind of ref it really was. See the *perlref* manpage.

Not a HASH reference
: (F) Perl was trying to evaluate a reference to a hash value, but found a reference to something else instead. You can use the *ref()* function to find out what kind of ref it really was. See the *perlref* manpage.

Not a perl script
: (F) The setuid emulator requires that scripts have a well-formed #! line even on machines that don't support the #! construct. The line must mention perl.

Not a SCALAR reference
: (F) Perl was trying to evaluate a reference to a scalar value, but found a reference to something else instead. You can use the *ref()* function to find out what kind of ref it really was. See the *perlref* manpage.

Not a subroutine reference
: (F) Perl was trying to evaluate a reference to a code value (that is, a subroutine), but found a reference to something else instead. You can use the *ref()* function to find out what kind of ref it really was. See also the *perlref* manpage.

Not a subroutine reference in overload table
: (F) An attempt was made to specify an entry in an overloading table that doesn't somehow point to a valid subroutine. See the *overload* manpage.

Not an ARRAY reference
: (F) Perl was trying to evaluate a reference to an array value, but found a reference to something else instead. You can use the *ref()* function to find out what kind of ref it really was. See the *perlref* manpage.

Not enough arguments for %s
: (F) The function requires more arguments than you specified.

Not enough format arguments
: (W) A format specified more picture fields than the next line supplied. See the *perlform* manpage.

Null filename used
: (F) You can't require the null filename, especially because on many machines that means the current directory! See the `require` entry in the *perlfunc* manpage.

Null picture in formline
: (F) The first argument to formline must be a valid format picture specification. It was found to be empty, which probably means you supplied it an uninitialized value. See the *perlform* manpage.

NULL OP IN RUN
: (P) Some internal routine called *run()* with a null opcode pointer.

Null realloc
: (P) An attempt was made to realloc NULL.

NULL regexp argument
: (P) The internal pattern matching routines blew it big time.

NULL regexp parameter
: (P) The internal pattern matching routines are out of their gourd.

Number too long
: (F) Perl limits the representation of decimal numbers in programs to about about 250 characters. You've exceeded that length. Future versions of Perl are likely to eliminate this arbitrary limitation. In the meantime, try using scientific notation (e.g. "1e6" instead of "1_000_000").

Odd number of elements in hash assignment
: (S) You specified an odd number of elements to initialize a hash, which is odd, because hashes come in key/value pairs.

Offset outside string
: (F) You tried to do a read/write/send/recv operation with an offset pointing outside the buffer. This is difficult to imagine. The sole exception to this is that `sysread()` ing past the buffer will extend the buffer and zero pad the new area.

oops: oopsAV
: (S) An internal warning that the grammar is screwed up.

oops: oopsHV
: (S) An internal warning that the grammar is screwed up.

Operation '%s': no method found, %s
: (F) An attempt was made to perform an overloaded operation for which no handler was defined. While some handlers can be autogenerated in terms of other handlers, there is no default handler for any operation, unless `fallback` overloading key is specified to be true. See the *overload* manpage.

Operator or semicolon missing before %s
: (S) You used a variable or subroutine call where the parser was expecting an operator. The parser has assumed you really meant to use an operator, but this is highly likely to be incorrect. For example, if you say "*foo *foo" it will be interpreted as if you said "*foo * 'foo'".

Out of memory for yacc stack
: (F) The yacc parser wanted to grow its stack so it could continue parsing, but *realloc()* wouldn't give it more memory, virtual or otherwise.

Out of memory during request for %s
: (X|F) The *malloc()* function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request.

  The request was judged to be small, so the possibility to trap it depends on the way perl was compiled. By default it is not trappable. However, if compiled for this, Perl may use the contents of `$^M` as an emergency pool after *die()*ing with this message. In this case the error is trappable *once*.

Out of memory during "large" request for %s
: (F) The *malloc()* function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request. However, the request was judged large enough (compile-time default is 64K), so a possibility to shut down by trapping this error is granted.

Out of memory during ridiculously large request
: (F) You can't allocate more than 2^31+"small amount" bytes. This error is most likely to be caused by a typo in the Perl program. e.g., `$arr[time]` instead of `$arr[$time]`.

page overflow
: (W) A single call to *write()* produced more lines than can fit on a page. See the *perlform* manpage.

panic: ck_grep
: (P) Failed an internal consistency check trying to compile a grep.

panic: ck_split
: (P) Failed an internal consistency check trying to compile a split.

panic: corrupt saved stack index
: (P) The savestack was requested to restore more localized values than there are in the savestack.

panic: die %s
: (P) We popped the context stack to an eval context, and then discovered it wasn't an eval context.

panic: do_match
: (P) The internal *pp_match()* routine was called with invalid operational data.

panic: do_split
: (P) Something terrible went wrong in setting up for the split.

panic: do_subst
: (P) The internal *pp_subst()* routine was called with invalid operational data.

panic: do_trans
: (P) The internal *do_trans()* routine was called with invalid operational data.

panic: frexp
: (P) The library function *frexp()* failed, making *printf*("%f") impossible.

panic: goto
: (P) We popped the context stack to a context with the specified label, and then discovered it wasn't a context we know how to do a goto in.

panic: INTERPCASEMOD
: (P) The lexer got into a bad state at a case modifier.

panic: INTERPCONCAT
: (P) The lexer got into a bad state parsing a string with brackets.

panic: last
: (P) We popped the context stack to a block context, and then discovered it wasn't a block context.

panic: leave_scope clearsv
: (P) A writable lexical variable became read-only somehow within the scope.

panic: leave_scope inconsistency
: (P) The savestack probably got out of sync. At least, there was an invalid enum on the top of it.

panic: malloc
: (P) Something requested a negative number of bytes of malloc.

panic: mapstart
: (P) The compiler is screwed up with respect to the *map()* function.

panic: null array
: (P) One of the internal array routines was passed a null AV pointer.

panic: pad_alloc
: (P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_free curpad
: (P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_free po
: (P) An invalid scratch pad offset was detected internally.

panic: pad_reset curpad
: (P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_sv po
(P) An invalid scratch pad offset was detected internally.

panic: pad_swipe curpad
(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_swipe po
(P) An invalid scratch pad offset was detected internally.

panic: pp_iter
(P) The foreach iterator got called in a non-loop context frame.

panic: realloc
(P) Something requested a negative number of bytes of realloc.

panic: restartop
(P) Some internal routine requested a goto (or something like it), and didn't supply the destination.

panic: return
(P) We popped the context stack to a subroutine or eval context, and then discovered it wasn't a subroutine or eval context.

panic: scan_num
(P) *scan_num()* got called on something that wasn't a number.

panic: sv_insert
(P) The *sv_insert()* routine was told to remove more string than there was string.

panic: top_env
(P) The compiler attempted to do a goto, or something weird like that.

panic: yylex
(P) The lexer got into a bad state while processing a case modifier.

Parentheses missing around "%s" list
(W) You said something like

```
my $foo, $bar = @_;
```

when you meant

```
my ($foo, $bar) = @_;
```

Remember that "my" and "local" bind closer than comma.

Perl %3.3f required—this is only version %s, stopped
(F) The module in question uses features of a version of Perl more recent than the currently running version. How long has it been since you upgraded, anyway? See the `require` entry in the *perlfunc* manpage.

Permission denied
(F) The setuid emulator in suidperl decided you were up to no good.

pid %d not a child
(W) A warning peculiar to VMS. *Waitpid()* was asked to wait for a process which isn't a subprocess of the current process. While this is fine from VMS' perspective, it's probably not what you intended.

POSIX getpgrp can't take an argument
(F) Your C compiler uses POSIX *getpgrp()*, which takes no argument, unlike the BSD version, which takes a pid.

Possible attempt to put comments in qw() list
(W) *qw()* lists contain items separated by whitespace; as with literal strings, comment characters are not ignored, but are instead treated as literal data. (You may have used different delimiters than the

parentheses shown here; braces are also frequently used.)

You probably wrote something like this:

```
@list = qw(
    a # a comment
    b # another comment
);
```

when you should have written this:

```
@list = qw(
    a
    b
);
```

If you really want comments, build your list the old-fashioned way, with quotes and commas:

```
@list = (
    'a',     # a comment
    'b',     # another comment
);
```

Possible attempt to separate words with commas

(W) *qw()* lists contain items separated by whitespace; therefore commas aren't needed to separate the items. (You may have used different delimiters than the parentheses shown here; braces are also frequently used.)

You probably wrote something like this:

```
qw! a, b, c !;
```

which puts literal commas into some of the list items. Write it without commas if you don't want them to appear in your data:

```
qw! a b c !;
```

Possible memory corruption: %s overflowed 3rd argument

(F) An *ioctl()* or *fcntl()* returned more than Perl was bargaining for. Perl guesses a reasonable buffer size, but puts a sentinel byte at the end of the buffer just in case. This sentinel byte got clobbered, and Perl assumes that memory is now corrupted. See the `ioctl` entry in the *perlfunc* manpage.

Precedence problem: open %s should be open(%s)

(S) The old irregular construct

```
open FOO || die;
```

is now misinterpreted as

```
open(FOO || die);
```

because of the strict regularization of Perl 5's grammar into unary and list operators. (The old open was a little of both.) You must put parentheses around the filehandle, or use the new "or" operator instead of "||".

print on closed filehandle %s

(W) The filehandle you're printing on got itself closed sometime before now. Check your logic flow.

printf on closed filehandle %s
(W) The filehandle you're writing to got itself closed sometime before now. Check your logic flow.

Probable precedence problem on %s
(W) The compiler found a bareword where it expected a conditional, which often indicates that an || or && was parsed as part of the last argument of the previous construct, for example:

```
open FOO || die;
```

Prototype mismatch: %s vs %s
(S) The subroutine being declared or defined had previously been declared or defined with a different function prototype.

Range iterator outside integer range
(F) One (or both) of the numeric arguments to the range operator ".." are outside the range which can be represented by integers internally. One possible workaround is to force Perl to use magical string increment by prepending "0" to your numbers.

Read on closed filehandle <%s>
(W) The filehandle you're reading from got itself closed sometime before now. Check your logic flow.

Reallocation too large: %lx
(F) You can't allocate more than 64K on an MS−DOS machine.

Recompile perl with **−D**DEBUGGING to use **−D** switch
(F) You can't use the **−D** option unless the code to produce the desired output is compiled into Perl, which entails some overhead, which is why it's currently left out of your copy.

Recursive inheritance detected in package '%s'
(F) More than 100 levels of inheritance were used. Probably indicates an unintended loop in your inheritance hierarchy.

Recursive inheritance detected while looking for method '%s' in package '%s'
(F) More than 100 levels of inheritance were encountered while invoking a method. Probably indicates an unintended loop in your inheritance hierarchy.

Reference found where even-sized list expected
(W) You gave a single reference where Perl was expecting a list with an even number of elements (for assignment to a hash). This usually means that you used the anon hash constructor when you meant to use parens. In any case, a hash requires key/value **pairs**.

```
%hash = { one => 1, two => 2, };     # WRONG
%hash = [ qw/ an anon array / ];     # WRONG
%hash = ( one => 1, two => 2, );     # right
%hash = qw( one 1 two 2 );                   # also fine
```

Reference miscount in sv_replace()
(W) The internal *sv_replace()* function was handed a new SV with a reference count of other than 1.

regexp *+ operand could be empty
(F) The part of the regexp subject to either the * or + quantifier could match an empty string.

regexp memory corruption
(P) The regular expression engine got confused by what the regular expression compiler gave it.

regexp out of space
(P) A "can't happen" error, because *safemalloc()* should have caught it earlier.

regexp too big
(F) The current implementation of regular expressions uses shorts as address offsets within a string. Unfortunately this means that if the regular expression compiles to longer than 32767, it'll blow up. Usually when you want a regular expression this big, there is a better way to do it with multiple

statements. See the *perlre* manpage.

Reversed %s= operator

(W) You wrote your assignment operator backwards. The = must always comes last, to avoid ambiguity with subsequent unary operators.

Runaway format

(F) Your format contained the ~~ repeat-until-blank sequence, but it produced 200 lines at once, and the 200th line looked exactly like the 199th line. Apparently you didn't arrange for the arguments to exhaust themselves, either by using ^ instead of @ (for scalar variables), or by shifting or popping (for array variables). See the *perlform* manpage.

Scalar value @%s[%s] better written as $%s[%s]

(W) You've used an array slice (indicated by @) to select a single element of an array. Generally it's better to ask for a scalar value (indicated by $). The difference is that `$foo[&bar]` always behaves like a scalar, both when assigning to it and when evaluating its argument, while `@foo[&bar]` behaves like a list when you assign to it, and provides a list context to its subscript, which can do weird things if you're expecting only one subscript.

On the other hand, if you were actually hoping to treat the array element as a list, you need to look into how references work, because Perl will not magically convert between scalars and lists for you. See the *perlref* manpage.

Scalar value @%s{%s} better written as $%s{%s}

(W) You've used a hash slice (indicated by @) to select a single element of a hash. Generally it's better to ask for a scalar value (indicated by $). The difference is that `$foo{&bar}` always behaves like a scalar, both when assigning to it and when evaluating its argument, while `@foo{&bar}` behaves like a list when you assign to it, and provides a list context to its subscript, which can do weird things if you're expecting only one subscript.

On the other hand, if you were actually hoping to treat the hash element as a list, you need to look into how references work, because Perl will not magically convert between scalars and lists for you. See the *perlref* manpage.

Script is not setuid/setgid in suidperl

(F) Oddly, the suidperl program was invoked on a script without a setuid or setgid bit set. This doesn't make much sense.

Search pattern not terminated

(F) The lexer couldn't find the final delimiter of a // or m{} construct. Remember that bracketing delimiters count nesting level. Missing the leading `$` from a variable `$m` may cause this error.

%sseek() on unopened file

(W) You tried to use the *seek()* or *sysseek()* function on a filehandle that was either never opened or has since been closed.

select not implemented

(F) This machine doesn't implement the *select()* system call.

sem%s not implemented

(F) You don't have System V semaphore IPC on your system.

semi-panic: attempt to dup freed string

(S) The internal *newSVsv()* routine was called to duplicate a scalar that had previously been marked as free.

Semicolon seems to be missing

(W) A nearby syntax error was probably caused by a missing semicolon, or possibly some other missing operator, such as a comma.

Send on closed socket

(W) The filehandle you're sending to got itself closed sometime before now. Check your logic flow.

Sequence (? incomplete

(F) A regular expression ended with an incomplete extension (?. See the *perlre* manpage.

Sequence (?#... not terminated

(F) A regular expression comment must be terminated by a closing parenthesis. Embedded parentheses aren't allowed. See the *perlre* manpage.

Sequence (?%s...) not implemented

(F) A proposed regular expression extension has the character reserved but has not yet been written. See the *perlre* manpage.

Sequence (?%s...) not recognized

(F) You used a regular expression extension that doesn't make sense. See the *perlre* manpage.

Server error

Also known as "500 Server error".

**This is a CGI error, not a Perl error**.

You need to make sure your script is executable, is accessible by the user CGI is running the script under (which is probably not the user account you tested it under), does not rely on any environment variables (like PATH) from the user it isn't running under, and isn't in a location where the CGI server can't find it, basically, more or less. Please see the following for more information:

```
http://www.perl.com/perl/faq/idiots-guide.html
http://www.perl.com/perl/faq/perl-cgi-faq.html
ftp://rtfm.mit.edu/pub/usenet/news.answers/www/cgi-faq
http://hoohoo.ncsa.uiuc.edu/cgi/interface.html
http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html
```

setegid() not implemented

(F) You tried to assign to `$)`, and your operating system doesn't support the *setegid()* system call (or equivalent), or at least Configure didn't think so.

seteuid() not implemented

(F) You tried to assign to `$>`, and your operating system doesn't support the *seteuid()* system call (or equivalent), or at least Configure didn't think so.

setrgid() not implemented

(F) You tried to assign to `$(`, and your operating system doesn't support the *setrgid()* system call (or equivalent), or at least Configure didn't think so.

setruid() not implemented

(F) You tried to assign to `$<`, and your operating system doesn't support the *setruid()* system call (or equivalent), or at least Configure didn't think so.

Setuid/gid script is writable by world

(F) The setuid emulator won't run a script that is writable by the world, because the world might have written on it already.

shm%s not implemented

(F) You don't have System V shared memory IPC on your system.

shutdown() on closed fd

(W) You tried to do a shutdown on a closed socket. Seems a bit superfluous.

SIG%s handler "%s" not defined

(W) The signal handler named in `%SIG` doesn't, in fact, exist. Perhaps you put it into the wrong package?

sort is now a reserved word

(F) An ancient error message that almost nobody ever runs into anymore. But before sort was a keyword, people sometimes used it as a filehandle.

Sort subroutine didn't return a numeric value

(F) A sort comparison routine must return a number. You probably blew it by not using <=> or `cmp`, or by not using them correctly. See the `sort` entry in the *perlfunc* manpage.

Sort subroutine didn't return single value

(F) A sort comparison subroutine may not return a list value with more or less than one element. See the `sort` entry in the *perlfunc* manpage.

Split loop

(P) The split was looping infinitely. (Obviously, a split shouldn't iterate more times than there are characters of input, which is what happened.) See the `split` entry in the *perlfunc* manpage.

Stat on unopened file <%s>

(W) You tried to use the *stat()* function (or an equivalent file test) on a filehandle that was either never opened or has since been closed.

Statement unlikely to be reached

(W) You did an *exec()* with some statement after it other than a *die()*. This is almost always an error, because *exec()* never returns unless there was a failure. You probably wanted to use *system()* instead, which does return. To suppress this warning, put the *exec()* in a block by itself.

Stub found while resolving method '%s' overloading '%s' in package '%s'

(P) Overloading resolution over `@ISA` tree may be broken by importation stubs. Stubs should never be implicitely created, but explicit calls to `can` may break this.

Subroutine %s redefined

(W) You redefined a subroutine. To suppress this warning, say

```
{
    local $^W = 0;
    eval "sub name { ... }";
}
```

Substitution loop

(P) The substitution was looping infinitely. (Obviously, a substitution shouldn't iterate more times than there are characters of input, which is what happened.) See the discussion of substitution in the section on *Quote and Quote-like Operators* in the *perlop* manpage.

Substitution pattern not terminated

(F) The lexer couldn't find the interior delimiter of a s/// or s{}{} construct. Remember that bracketing delimiters count nesting level. Missing the leading `$` from variable `$s` may cause this error.

Substitution replacement not terminated

(F) The lexer couldn't find the final delimiter of a s/// or s{}{} construct. Remember that bracketing delimiters count nesting level. Missing the leading `$` from variable `$s` may cause this error.

substr outside of string

(S),(W) You tried to reference a *substr()* that pointed outside of a string. That is, the absolute value of the offset was larger than the length of the string. See the `substr` entry in the *perlfunc* manpage. This warning is mandatory if substr is used in an lvalue context (as the left hand side of an assignment or as a subroutine argument for example).

suidperl is no longer needed since %s

(F) Your Perl was compiled with **−D**SETUID_SCRIPTS_ARE_SECURE_NOW, but a version of the setuid emulator somehow got run anyway.

syntax error

(F) Probably means you had a syntax error. Common reasons include:

```
A keyword is misspelled.
A semicolon is missing.
A comma is missing.
An opening or closing parenthesis is missing.
An opening or closing brace is missing.
A closing quote is missing.
```

Often there will be another error message associated with the syntax error giving more information. (Sometimes it helps to turn on **−w**.) The error message itself often tells you where it was in the line when it decided to give up. Sometimes the actual error is several tokens before this, because Perl is good at understanding random input. Occasionally the line number may be misleading, and once in a blue moon the only way to figure out what's triggering the error is to call `perl -c` repeatedly, chopping away half the program each time to see if the error went away. Sort of the cybernetic version of 20 questions.

syntax error at line %d: '%s' unexpected

(A) You've accidentally run your script through the Bourne shell instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

System V %s is not implemented on this machine

(F) You tried to do something with a function beginning with "sem", "shm", or "msg" but that System V IPC is not implemented in your machine. In some machines the functionality can exist but be unconfigured. Consult your system support.

Syswrite on closed filehandle

(W) The filehandle you're writing to got itself closed sometime before now. Check your logic flow.

Target of goto is too deeply nested

(F) You tried to use `goto` to reach a label that was too deeply nested for Perl to reach. Perl is doing you a favor by refusing.

tell() on unopened file

(W) You tried to use the *tell()* function on a filehandle that was either never opened or has since been closed.

Test on unopened file <%s>

(W) You tried to invoke a file test operator on a filehandle that isn't open. Check your logic. See also the section on *−X* in the *perlfunc* manpage.

That use of $[ is unsupported

(F) Assignment to `$[` is now strictly circumscribed, and interpreted as a compiler directive. You may say only one of

```
$[ = 0;
$[ = 1;
...
local $[ = 0;
local $[ = 1;
...
```

This is to prevent the problem of one module changing the array base out from under another module inadvertently. See the section on *$[* in the *perlvar* manpage.

The %s function is unimplemented

The function indicated isn't implemented on this architecture, according to the probings of Configure.

The crypt() function is unimplemented due to excessive paranoia

(F) Configure couldn't find the *crypt()* function on your machine, probably because your vendor didn't supply it, probably because they think the U.S. Government thinks it's a secret, or at least that they will continue to pretend that it is. And if you quote me on that, I will deny it.

The stat preceding `-l _` wasn't an lstat

(F) It makes no sense to test the current stat buffer for symbolic linkhood if the last stat that wrote to the stat buffer already went past the symlink to get to the real file. Use an actual filename instead.

times not implemented

(F) Your version of the C library apparently doesn't do *times()*. I suspect you're not running on Unix.

Too few args to syscall

(F) There has to be at least one argument to *syscall()* to specify the system call to call, silly dilly.

Too late for "**–T**" option

(X) The #! line (or local equivalent) in a Perl script contains the **–T** option, but Perl was not invoked with **–T** in its command line. This is an error because, by the time Perl discovers a **–T** in a script, it's too late to properly taint everything from the environment. So Perl gives up.

If the Perl script is being executed as a command using the #! mechanism (or its local equivalent), this error can usually be fixed by editing the #! line so that the **–T** option is a part of Perl's first argument: e.g. change `perl -n -T` to `perl -T -n`.

If the Perl script is being executed as `perl scriptname`, then the **–T** option must appear on the command line: `perl -T scriptname`.

Too late for "–%s" option

(X) The #! line (or local equivalent) in a Perl script contains the **–M** or **–m** option. This is an error because **–M** and **–m** options are not intended for use inside scripts. Use the `use` pragma instead.

Too many ('s

Too many )'s

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

Too many args to syscall

(F) Perl supports a maximum of only 14 args to *syscall()*.

Too many arguments for %s

(F) The function requires fewer arguments than you specified.

trailing \ in regexp

(F) The regular expression ends with an unbackslashed backslash. Backslash it. See the *perlre* manpage.

Transliteration pattern not terminated

(F) The lexer couldn't find the interior delimiter of a tr/// or tr[][] or y/// or y[][] construct. Missing the leading `$` from variables `$tr` or `$y` may cause this error.

Transliteration replacement not terminated

(F) The lexer couldn't find the final delimiter of a tr/// or tr[][] construct.

truncate not implemented

(F) Your machine doesn't implement a file truncation mechanism that Configure knows about.

Type of arg %d to %s must be %s (not %s)

(F) This function requires the argument in that position to be of a certain type. Arrays must be `@NAME` or `@{EXPR}`. Hashes must be `%NAME` or `%{EXPR}`. No implicit dereferencing is allowed— use the {EXPR} forms as an explicit dereference. See the *perlref* manpage.

umask: argument is missing initial 0
: (W) A umask of 222 is incorrect. It should be 0222, because octal literals always start with 0 in Perl, as in C.

umask not implemented
: (F) Your machine doesn't implement the umask function and you tried to use it to restrict permissions for yourself (EXPR & 0700).

Unable to create sub named "%s"
: (F) You attempted to create or access a subroutine with an illegal name.

Unbalanced context: %d more PUSHes than POPs
: (W) The exit code detected an internal inconsistency in how many execution contexts were entered and left.

Unbalanced saves: %d more saves than restores
: (W) The exit code detected an internal inconsistency in how many values were temporarily localized.

Unbalanced scopes: %d more ENTERs than LEAVEs
: (W) The exit code detected an internal inconsistency in how many blocks were entered and left.

Unbalanced tmps: %d more allocs than frees
: (W) The exit code detected an internal inconsistency in how many mortal scalars were allocated and freed.

Undefined format "%s" called
: (F) The format indicated doesn't seem to exist. Perhaps it's really in another package? See the *perlform* manpage.

Undefined sort subroutine "%s" called
: (F) The sort comparison routine specified doesn't seem to exist. Perhaps it's in a different package? See the `sort` entry in the *perlfunc* manpage.

Undefined subroutine &%s called
: (F) The subroutine indicated hasn't been defined, or if it was, it has since been undefined.

Undefined subroutine called
: (F) The anonymous subroutine you're trying to call hasn't been defined, or if it was, it has since been undefined.

Undefined subroutine in sort
: (F) The sort comparison routine specified is declared but doesn't seem to have been defined yet. See the `sort` entry in the *perlfunc* manpage.

Undefined top format "%s" called
: (F) The format indicated doesn't seem to exist. Perhaps it's really in another package? See the *perlform* manpage.

Undefined value assigned to typeglob
: (W) An undefined value was assigned to a typeglob, a la `*foo = undef`. This does nothing. It's possible that you really mean `undef *foo`.

unexec of %s into %s failed!
: (F) The *unexec()* routine failed for some reason. See your local FSF representative, who probably put it there in the first place.

Unknown BYTEORDER
: (F) There are no byte-swapping functions for a machine with this byte order.

unmatched () in regexp
: (F) Unbackslashed parentheses must always be balanced in regular expressions. If you're a vi user, the % key is valuable for finding the matching parenthesis. See the *perlre* manpage.

Unmatched right bracket

(F) The lexer counted more closing curly brackets (braces) than opening ones, so you're probably missing an opening bracket. As a general rule, you'll find the missing one (so to speak) near the place you were last editing.

unmatched [] in regexp

(F) The brackets around a character class must match. If you wish to include a closing bracket in a character class, backslash it or put it first. See the *perlre* manpage.

Unquoted string "%s" may clash with future reserved word

(W) You used a bareword that might someday be claimed as a reserved word. It's best to put such a word in quotes, or capitalize it somehow, or insert an underbar into it. You might also declare it as a subroutine.

Unrecognized character %s

(F) The Perl parser has no idea what to do with the specified character in your Perl script (or eval). Perhaps you tried to run a compressed script, a binary program, or a directory as a Perl program.

Unrecognized signal name "%s"

(F) You specified a signal name to the *kill()* function that was not recognized. Say `kill -l` in your shell to see the valid signal names on your system.

Unrecognized switch: −%s (−h will show valid options)

(F) You specified an illegal option to Perl. Don't do that. (If you think you didn't do that, check the #! line to see if it's supplying the bad switch on your behalf.)

Unsuccessful %s on filename containing newline

(W) A file operation was attempted on a filename, and that operation failed, PROBABLY because the filename contained a newline, PROBABLY because you forgot to *chop()* or *chomp()* it off. See the `chomp` entry in the *perlfunc* manpage.

Unsupported directory function "%s" called

(F) Your machine doesn't support *opendir()* and *readdir()*.

Unsupported function fork

(F) Your version of executable does not support forking.

Note that under some systems, like OS/2, there may be different flavors of Perl executables, some of which may support fork, some not. Try changing the name you call Perl by to `perl_`, `perl__`, and so on.

Unsupported function %s

(F) This machine doesn't implement the indicated function, apparently. At least, Configure doesn't think so.

Unsupported socket function "%s" called

(F) Your machine doesn't support the Berkeley socket mechanism, or at least that's what Configure thought.

Unterminated <> operator

(F) The lexer saw a left angle bracket in a place where it was expecting a term, so it's looking for the corresponding right angle bracket, and not finding it. Chances are you left some needed parentheses out earlier in the line, and you really meant a "less than".

Use of "$$<digit>" to mean "${$}<digit>" is deprecated

(D) Perl versions before 5.004 misinterpreted any type marker followed by "$" and a digit. For example, "$$0" was incorrectly taken to mean "${$}0" instead of "${$0}". This bug is (mostly) fixed in Perl 5.004.

However, the developers of Perl 5.004 could not fix this bug completely, because at least two widely-used modules depend on the old meaning of "$$0" in a string. So Perl 5.004 still interprets "$$<digit>" in the old (broken) way inside strings; but it generates this message as a warning. And in Perl 5.005, this special treatment will cease.

Use of $# is deprecated

(D) This was an ill-advised attempt to emulate a poorly defined **awk** feature. Use an explicit *printf()* or *sprintf()* instead.

Use of $* is deprecated

(D) This variable magically turned on multi-line pattern matching, both for you and for any luckless subroutine that you happen to call. You should use the new `//m` and `//s` modifiers now to do that without the dangerous action-at-a-distance effects of `$*`.

Use of %s in printf format not supported

(F) You attempted to use a feature of printf that is accessible from only C. This usually means there's a better way to do it in Perl.

Use of bare << to mean <<"" is deprecated

(D) You are now encouraged to use the explicitly quoted form if you wish to use an empty line as the terminator of the here-document.

Use of implicit split to @_ is deprecated

(D) It makes a lot of work for the compiler when you clobber a subroutine's argument list, so it's better if you assign the results of a *split()* explicitly to an array (or list).

Use of inherited AUTOLOAD for non-method %s() is deprecated

(D) As an (ahem) accidental feature, `AUTOLOAD` subroutines are looked up as methods (using the `@ISA` hierarchy) even when the subroutines to be autoloaded were called as plain functions (e.g. `Foo::bar()`), not as methods (e.g. `Foo->bar()` or `$obj->bar()`).

This bug will be rectified in Perl 5.005, which will use method lookup only for methods' `AUTOLOAD`s. However, there is a significant base of existing code that may be using the old behavior. So, as an interim step, Perl 5.004 issues an optional warning when non-methods use inherited `AUTOLOAD`s.

The simple rule is: Inheritance will not work when autoloading non-methods. The simple fix for old code is: In any module that used to depend on inheriting `AUTOLOAD` for non-methods from a base class named `BaseClass`, execute `*AUTOLOAD = \&BaseClass::AUTOLOAD` during startup.

In code that currently says `use AutoLoader; @ISA = qw(AutoLoader);` you should remove AutoLoader from `@ISA` and change `use AutoLoader;` to `use AutoLoader 'AUTOLOAD';`.

Use of reserved word "%s" is deprecated

(D) The indicated bareword is a reserved word. Future versions of perl may use it as a keyword, so you're better off either explicitly quoting the word in a manner appropriate for its context of use, or using a different name altogether. The warning can be suppressed for subroutine names by either adding a `&` prefix, or using a package qualifier, e.g. `&our()`, or `Foo::our()`.

Use of %s is deprecated

(D) The construct indicated is no longer recommended for use, generally because there's a better way to do it, and also because the old way has bad side effects.

Use of uninitialized value

(W) An undefined value was used as if it were already defined. It was interpreted as a "" or a 0, but maybe it was a mistake. To suppress this warning assign an initial value to your variables.

Useless use of "re" pragma

(W) You did `use re;` without any arguments. That isn't very useful.

Useless use of %s in void context

(W) You did something without a side effect in a context that does nothing with the return value, such as a statement that doesn't return a value from a block, or the left side of a scalar comma operator. Very often this points not to stupidity on your part, but a failure of Perl to parse your program the way you thought it would. For example, you'd get this if you mixed up your C precedence with Python precedence and said

```
$one, $two = 1, 2;
```

when you meant to say

```
($one, $two) = (1, 2);
```

Another common error is to use ordinary parentheses to construct a list reference when you should be using square or curly brackets, for example, if you say

```
$array = (1,2);
```

when you should have said

```
$array = [1,2];
```

The square brackets explicitly turn a list value into a scalar value, while parentheses do not. So when a parenthesized list is evaluated in a scalar context, the comma is treated like C's comma operator, which throws away the left argument, which is not what you want. See the *perlref* manpage for more on this.

untie attempted while %d inner references still exist
: (W) A copy of the object returned from `tie` (or `tied`) was still valid when `untie` was called.

Value of %s can be "0"; test with defined()
: (W) In a conditional expression, you used <HANDLE>, <*> (glob), `each()`, or `readdir()` as a boolean value. Each of these constructs can return a value of "0"; that would make the conditional expression false, which is probably not what you intended. When using these constructs in conditional expressions, test their values with the `defined` operator.

Variable "%s" is not imported%s
: (F) While "use strict" in effect, you referred to a global variable that you apparently thought was imported from another module, because something else of the same name (usually a subroutine) is exported by that module. It usually means you put the wrong funny character on the front of your variable.

Variable "%s" may be unavailable
: (W) An inner (nested) *anonymous* subroutine is inside a *named* subroutine, and outside that is another subroutine; and the anonymous (innermost) subroutine is referencing a lexical variable defined in the outermost subroutine. For example:

```
sub outermost { my $a; sub middle { sub { $a } } }
```

If the anonymous subroutine is called or referenced (directly or indirectly) from the outermost subroutine, it will share the variable as you would expect. But if the anonymous subroutine is called or referenced when the outermost subroutine is not active, it will see the value of the shared variable as it was before and during the *first* call to the outermost subroutine, which is probably not what you want.

In these circumstances, it is usually best to make the middle subroutine anonymous, using the `sub {}` syntax. Perl has specific support for shared variables in nested anonymous subroutines; a named subroutine in between interferes with this feature.

Variable "%s" will not stay shared
: (W) An inner (nested) *named* subroutine is referencing a lexical variable defined in an outer subroutine.

When the inner subroutine is called, it will probably see the value of the outer subroutine's variable as it was before and during the *first* call to the outer subroutine; in this case, after the first call to the outer subroutine is complete, the inner and outer subroutines will no longer share a common value for the variable. In other words, the variable will no longer be shared.

Furthermore, if the outer subroutine is anonymous and references a lexical variable outside itself, then the outer and inner subroutines will *never* share the given variable.

This problem can usually be solved by making the inner subroutine anonymous, using the `sub {}` syntax. When inner anonymous subs that reference variables in outer subroutines are called or referenced, they are automatically rebound to the current values of such variables.

Variable syntax

(A) You've accidentally run your script through **csh** instead of Perl. Check the #! line, or manually feed your script into Perl yourself.

perl: warning: Setting locale failed.

(S) The whole warning message will look something like:

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
        LC_ALL = "En_US",
        LANG = (unset)
    are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

Exactly what were the failed locale settings varies. In the above the settings were that the LC_ALL was "En_US" and the LANG had no value. This error means that Perl detected that you and/or your system administrator have set up the so-called variable system but Perl could not use those settings. This was not dead serious, fortunately: there is a "default locale" called "C" that Perl can and will use, the script will be run. Before you really fix the problem, however, you will get the same error message each time you run Perl. How to really fix the problem can be found in the *perllocale* manpage section **LOCALE PROBLEMS**.

Warning: something's wrong

(W) You passed *warn()* an empty string (the equivalent of `warn ""`) or you called it with no args and `$_` was empty.

Warning: unable to close filehandle %s properly

(S) The implicit *close()* done by an *open()* got an error indication on the *close()*. This usually indicates your file system ran out of disk space.

Warning: Use of "%s" without parentheses is ambiguous

(S) You wrote a unary operator followed by something that looks like a binary operator that could also have been interpreted as a term or unary operator. For instance, if you know that the rand function has a default argument of 1.0, and you write

```
rand + 5;
```

you may THINK you wrote the same thing as

```
rand() + 5;
```

but in actual fact, you got

```
rand(+5);
```

So put in parentheses to say what you really mean.

Write on closed filehandle

(W) The filehandle you're writing to got itself closed sometime before now. Check your logic flow.

X outside of string

(F) You had a pack template that specified a relative position before the beginning of the string being unpacked. See the `pack` entry in the *perlfunc* manpage.

x outside of string
: (F) You had a pack template that specified a relative position after the end of the string being unpacked. See the `pack` entry in the *perlfunc* manpage.

Xsub "%s" called in sort
: (F) The use of an external subroutine as a sort comparison is not yet supported.

Xsub called in sort
: (F) The use of an external subroutine as a sort comparison is not yet supported.

You can't use `-l` on a filehandle
: (F) A filehandle represents an opened file, and when you opened the file it already went past any symlink you are presumably trying to look for. Use a filename instead.

YOU HAVEN'T DISABLED SET-ID SCRIPTS IN THE KERNEL YET!
: (F) And you probably never will, because you probably don't have the sources to your kernel, and your vendor probably doesn't give a rip about what you want. Your best bet is to use the wrapsuid script in the eg directory to put a setuid C wrapper around your script.

You need to quote "%s"
: (W) You assigned a bareword as a signal handler name. Unfortunately, you already have a subroutine of that name declared, which means that Perl 5 will try to call the subroutine when the assignment is executed, which is probably not what you want. (If it IS what you want, put an & in front.)

[gs]etsockopt() on closed fd
: (W) You tried to get or set a socket option on a closed socket. Did you forget to check the return value of your *socket()* call? See the `getsockopt` entry in the *perlfunc* manpage.

\1 better written as $1
: (W) Outside of patterns, backreferences live on as variables. The use of backslashes is grandfathered on the right-hand side of a substitution, but stylistically it's better to use the variable form because other Perl programmers will expect it, and it works better if there are more than 9 backreferences.

'|' and '<' may not both be specified on command line
: (F) An error peculiar to VMS. Perl does its own command line redirection, and found that STDIN was a pipe, and that you also tried to redirect STDIN using '<'. Only one STDIN stream to a customer, please.

'|' and '>' may not both be specified on command line
: (F) An error peculiar to VMS. Perl does its own command line redirection, and thinks you tried to redirect stdout both to a file and into a pipe to another command. You need to choose one or the other, though nothing's stopping you from piping into a program or Perl script which 'splits' output into two streams, such as

```
open(OUT,">$ARGV[0]") or die "Can't write to $ARGV[0]: $!";
while (<STDIN>) {
    print;
    print OUT;
}
close OUT;
```

Got an error from DosAllocMem
: (P) An error peculiar to OS/2. Most probably you're using an obsolete version of Perl, and this should not happen anyway.

Malformed PERLLIB_PREFIX
: (F) An error peculiar to OS/2. PERLLIB_PREFIX should be of the form

```
prefix1;prefix2
```

or

```
prefix1 prefix2
```

with nonempty prefix1 and prefix2. If `prefix1` is indeed a prefix of a builtin library search path, prefix2 is substituted. The error may appear if components are not found, or are too long. See "PERLLIB_PREFIX" in *README.os2*.

PERL_SH_DIR too long
: (F) An error peculiar to OS/2. PERL_SH_DIR is the directory to find the `sh`−shell in. See "PERL_SH_DIR" in *README.os2*.

Process terminated by SIG%s
: (W) This is a standard message issued by OS/2 applications, while *nix applications die in silence. It is considered a feature of the OS/2 port. One can easily disable this by appropriate sighandlers, see the section on *Signals* in the *perlipc* manpage. See also "Process terminated by SIGTERM/SIGINT" in *README.os2*.

**NAME**

perlsec – Perl security

**DESCRIPTION**

Perl is designed to make it easy to program securely even when running with extra privileges, like setuid or setgid programs. Unlike most command line shells, which are based on multiple substitution passes on each line of the script, Perl uses a more conventional evaluation scheme with fewer hidden snags. Additionally, because the language has more builtin functionality, it can rely less upon external (and possibly untrustworthy) programs to accomplish its purposes.

Perl automatically enables a set of special security checks, called *taint mode*, when it detects its program running with differing real and effective user or group IDs. The setuid bit in Unix permissions is mode 04000, the setgid bit mode 02000; either or both may be set. You can also enable taint mode explicitly by using the **–T** command line flag. This flag is *strongly* suggested for server programs and any program run on behalf of someone else, such as a CGI script. Once taint mode is on, it's on for the remainder of your script.

While in this mode, Perl takes special precautions called *taint checks* to prevent both obvious and subtle traps. Some of these checks are reasonably simple, such as verifying that path directories aren't writable by others; careful programmers have always used checks like these. Other checks, however, are best supported by the language itself, and it is these checks especially that contribute to making a set-id Perl program more secure than the corresponding C program.

You may not use data derived from outside your program to affect something else outside your program—at least, not by accident. All command line arguments, environment variables, locale information (see the *perllocale* manpage), results of certain system calls (readdir, readlink, the gecos field of getpw* calls), and all file input are marked as "tainted". Tainted data may not be used directly or indirectly in any command that invokes a sub-shell, nor in any command that modifies files, directories, or processes. (**Important exception**: If you pass a list of arguments to either `system` or `exec`, the elements of that list are **NOT** checked for taintedness.) Any variable set to a value derived from tainted data will itself be tainted, even if it is logically impossible for the tainted data to alter the variable. Because taintedness is associated with each scalar value, some elements of an array can be tainted and others not.

For example:

```
$arg = shift;               # $arg is tainted
$hid = $arg, 'bar';         # $hid is also tainted
$line = <>;                 # Tainted
$line = <STDIN>;            # Also tainted
open FOO, "/home/me/bar" or die $!;
$line = <FOO>;              # Still tainted
$path = $ENV{'PATH'};       # Tainted, but see below
$data = 'abc';              # Not tainted

system "echo $arg";         # Insecure
system "/bin/echo", $arg;   # Secure (doesn't use sh)
system "echo $hid";         # Insecure
system "echo $data";        # Insecure until PATH set

$path = $ENV{'PATH'};       # $path now tainted

$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

$path = $ENV{'PATH'};       # $path now NOT tainted
system "echo $data";        # Is secure now!
```

```
open(FOO, "< $arg");          # OK - read-only file
open(FOO, "> $arg");          # Not OK - trying to write

open(FOO,"echo $arg|");       # Not OK, but...
open(FOO,"-|")
    or exec 'echo', $arg;     # OK

$shout = `echo $arg`;         # Insecure, $shout now tainted

unlink $data, $arg;           # Insecure
umask $arg;                   # Insecure

exec "echo $arg";             # Insecure
exec "echo", $arg;            # Secure (doesn't use the shell)
exec "sh", '-c', $arg;        # Considered secure, alas!

@files = <*.c>;               # Always insecure (uses csh)
@files = glob('*.c');         # Always insecure (uses csh)
```

If you try to do something insecure, you will get a fatal error saying something like "Insecure dependency" or "Insecure `$ENV{PATH}`". Note that you can still write an insecure **system** or **exec**, but only by explicitly doing something like the "considered secure" example above.

### Laundering and Detecting Tainted Data

To test whether a variable contains tainted data, and whose use would thus trigger an "Insecure dependency" message, check your nearby CPAN mirror for the *Taint.pm* module, which should become available around November 1997. Or you may be able to use the following *is_tainted()* function.

```
sub is_tainted {
    return ! eval {
        join('',@_), kill 0;
        1;
    };
}
```

This function makes use of the fact that the presence of tainted data anywhere within an expression renders the entire expression tainted. It would be inefficient for every operator to test every argument for taintedness. Instead, the slightly more efficient and conservative approach is used that if any tainted value has been accessed within the same expression, the whole expression is considered tainted.

But testing for taintedness gets you only so far. Sometimes you have just to clear your data's taintedness. The only way to bypass the tainting mechanism is by referencing subpatterns from a regular expression match. Perl presumes that if you reference a substring using `$1`, `$2`, etc., that you knew what you were doing when you wrote the pattern. That means using a bit of thought— don't just blindly untaint anything, or you defeat the entire mechanism. It's better to verify that the variable has only good characters (for certain values of "good") rather than checking whether it has any bad characters. That's because it's far too easy to miss bad characters that you never thought of.

Here's a test to make sure that the data contains nothing but "word" characters (alphabetics, numerics, and underscores), a hyphen, an at sign, or a dot.

```
if ($data =~ /^([-\@\w.]+)$/) {
    $data = $1;                     # $data now untainted
} else {
    die "Bad data in $data";        # log this somewhere
}
```

This is fairly secure because `/\w+/` doesn't normally match shell metacharacters, nor are dot, dash, or at going to mean something special to the shell. Use of `/.+/` would have been insecure in theory because it lets everything through, but Perl doesn't check for that. The lesson is that when untainting, you must be exceedingly careful with your patterns. Laundering data using regular expression is the *ONLY* mechanism for untainting dirty data, unless you use the strategy detailed below to fork a child of lesser privilege.

The example does not untaint `$data` if `use locale` is in effect, because the characters matched by `\w` are determined by the locale. Perl considers that locale definitions are untrustworthy because they contain data from outside the program. If you are writing a locale-aware program, and want to launder data with a regular expression containing `\w`, put `no locale` ahead of the expression in the same block. See the `SECURITY` entry in the *perllocale* manpage for further discussion and examples.

### Switches On the "#!" Line

When you make a script executable, in order to make it usable as a command, the system will pass switches to perl from the script's #! line. Perl checks that any command line switches given to a setuid (or setgid) script actually match the ones set on the #! line. Some Unix and Unix-like environments impose a one-switch limit on the #! line, so you may need to use something like `-wU` instead of `-w -U` under such systems. (This issue should arise only in Unix or Unix-like environments that support #! and setuid or setgid scripts.)

### Cleaning Up Your Path

For "Insecure `$ENV{PATH}`" messages, you need to set `$ENV{'PATH'}` to a known value, and each directory in the path must be non-writable by others than its owner and group. You may be surprised to get this message even if the pathname to your executable is fully qualified. This is *not* generated because you didn't supply a full path to the program; instead, it's generated because you never set your PATH environment variable, or you didn't set it to something that was safe. Because Perl can't guarantee that the executable in question isn't itself going to turn around and execute some other program that is dependent on your PATH, it makes sure you set the PATH.

The PATH isn't the only environment variable which can cause problems. Because some shells may use the variables IFS, CDPATH, ENV, and BASH_ENV, Perl checks that those are either empty or untainted when starting subprocesses. You may wish to add something like this to your setid and taint-checking scripts.

```
    delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};   # Make %ENV safer
```

It's also possible to get into trouble with other operations that don't care whether they use tainted values. Make judicious use of the file tests in dealing with any user-supplied filenames. When possible, do opens and such **after** properly dropping any special user (or group!) privileges. Perl doesn't prevent you from opening tainted filenames for reading, so be careful what you print out. The tainting mechanism is intended to prevent stupid mistakes, not to remove the need for thought.

Perl does not call the shell to expand wild cards when you pass **system** and **exec** explicit parameter lists instead of strings with possible shell wildcards in them. Unfortunately, the **open**, **glob**, and backtick functions provide no such alternate calling convention, so more subterfuge will be required.

Perl provides a reasonably safe way to open a file or pipe from a setuid or setgid program: just create a child process with reduced privilege who does the dirty work for you. First, fork a child using the special **open** syntax that connects the parent and child by a pipe. Now the child resets its ID set and any other per-process attributes, like environment variables, umasks, current working directories, back to the originals or known safe values. Then the child process, which no longer has any special permissions, does the **open** or other system call. Finally, the child passes the data it managed to access back to the parent. Because the file or pipe was opened in the child while running under less privilege than the parent, it's not apt to be tricked into doing something it shouldn't.

Here's a way to do backticks reasonably safely. Notice how the **exec** is not called with a string that the shell could expand. This is by far the best way to call something that might be subjected to shell escapes: just never call the shell at all.

```
use English;
die "Can't fork: $!" unless defined $pid = open(KID, "-|");
if ($pid) {           # parent
    while (<KID>) {
        # do something
    }
    close KID;
} else {
    my @temp = ($EUID, $EGID);
    $EUID = $UID;
    $EGID = $GID;    #      initgroups() also called!
    # Make sure privs are really gone
    ($EUID, $EGID) = @temp;
    die "Can't drop privileges"
            unless $UID == $EUID  && $GID eq $EGID;
    $ENV{PATH} = "/bin:/usr/bin";
    exec 'myprog', 'arg1', 'arg2'
        or die "can't exec myprog: $!";
}
```

A similar strategy would work for wildcard expansion via `glob`, although you can use `readdir` instead.

Taint checking is most useful when although you trust yourself not to have written a program to give away the farm, you don't necessarily trust those who end up using it not to try to trick it into doing something bad. This is the kind of security checking that's useful for set-id programs and programs launched on someone else's behalf, like CGI programs.

This is quite different, however, from not even trusting the writer of the code not to try to do something evil. That's the kind of trust needed when someone hands you a program you've never seen before and says, "Here, run this." For that kind of safety, check out the Safe module, included standard in the Perl distribution. This module allows the programmer to set up special compartments in which all system operations are trapped and namespace access is carefully controlled.

### Security Bugs

Beyond the obvious problems that stem from giving special privileges to systems as flexible as scripts, on many versions of Unix, set-id scripts are inherently insecure right from the start. The problem is a race condition in the kernel. Between the time the kernel opens the file to see which interpreter to run and when the (now-set-id) interpreter turns around and reopens the file to interpret it, the file in question may have changed, especially if you have symbolic links on your system.

Fortunately, sometimes this kernel "feature" can be disabled. Unfortunately, there are two ways to disable it. The system can simply outlaw scripts with any set-id bit set, which doesn't help much. Alternately, it can simply ignore the set-id bits on scripts. If the latter is true, Perl can emulate the setuid and setgid mechanism when it notices the otherwise useless setuid/gid bits on Perl scripts. It does this via a special executable called **suidperl** that is automatically invoked for you if it's needed.

However, if the kernel set-id script feature isn't disabled, Perl will complain loudly that your set-id script is insecure. You'll need to either disable the kernel set-id script feature, or put a C wrapper around the script. A C wrapper is just a compiled program that does nothing except call your Perl program. Compiled programs are not subject to the kernel bug that plagues set-id scripts. Here's a simple wrapper, written in C:

```
#define REAL_PATH "/path/to/script"
main(ac, av)
    char **av;
{
    execv(REAL_PATH, av);
}
```

Compile this wrapper into a binary executable and then make *it* rather than your script setuid or setgid.

See the program **wrapsuid** in the *eg* directory of your Perl distribution for a convenient way to do this automatically for all your setuid Perl programs. It moves setuid scripts into files with the same name plus a leading dot, and then compiles a wrapper like the one above for each of them.

In recent years, vendors have begun to supply systems free of this inherent security bug. On such systems, when the kernel passes the name of the set-id script to open to the interpreter, rather than using a pathname subject to meddling, it instead passes */dev/fd/3*. This is a special file already opened on the script, so that there can be no race condition for evil scripts to exploit. On these systems, Perl should be compiled with `-DSETUID_SCRIPTS_ARE_SECURE_NOW`. The **Configure** program that builds Perl tries to figure this out for itself, so you should never have to specify this yourself. Most modern releases of SysVr4 and BSD 4.4 use this approach to avoid the kernel race condition.

Prior to release 5.003 of Perl, a bug in the code of **suidperl** could introduce a security hole in systems compiled with strict POSIX compliance.

### Protecting Your Programs

There are a number of ways to hide the source to your Perl programs, with varying levels of ‘‘security’’.

First of all, however, you *can't* take away read permission, because the source code has to be readable in order to be compiled and interpreted. (That doesn't mean that a CGI script's source is readable by people on the web, though.) So you have to leave the permissions at the socially friendly 0755 level. This lets people on your local system only see your source.

Some people mistakenly regard this as a security problem. If your program does insecure things, and relies on people not knowing how to exploit those insecurities, it is not secure. It is often possible for someone to determine the insecure things and exploit them without viewing the source. Security through obscurity, the name for hiding your bugs instead of fixing them, is little security indeed.

You can try using encryption via source filters (Filter::* from CPAN). But crackers might be able to decrypt it. You can try using the byte code compiler and interpreter described below, but crackers might be able to de-compile it. You can try using the native-code compiler described below, but crackers might be able to disassemble it. These pose varying degrees of difficulty to people wanting to get at your code, but none can definitively conceal it (this is true of every language, not just Perl).

If you're concerned about people profiting from your code, then the bottom line is that nothing but a restrictive licence will give you legal security. License your software and pepper it with threatening statements like ‘‘This is unpublished proprietary software of XYZ Corp. Your access to it does not give you permission to use it blah blah blah.’’ You should see a lawyer to be sure your licence's wording will stand up in court.

## SEE ALSO

the *perlrun* manpage for its description of cleaning up environment variables.

## NAME

perltrap – Perl traps for the unwary

## DESCRIPTION

The biggest trap of all is forgetting to use the **–w** switch; see the *perlrun* manpage. The second biggest trap is not making your entire program runnable under `use strict`. The third biggest trap is not reading the list of changes in this version of Perl; see the *perldelta* manpage.

### Awk Traps

Accustomed **awk** users should take special note of the following:

- The English module, loaded via

  ```
  use English;
  ```

  allows you to refer to special variables (like `$/`) with names (like `$RS`), as though they were in **awk**; see the *perlvar* manpage for details.
- Semicolons are required after all simple statements in Perl (except at the end of a block). Newline is not a statement delimiter.
- Curly brackets are required on `if`s and `while`s.
- Variables begin with “$”, “@” or “%” in Perl.
- Arrays index from 0. Likewise string positions in *substr()* and *index()*.
- You have to decide whether your array has numeric or string indices.
- Hash values do not spring into existence upon mere reference.
- You have to decide whether you want to use string or numeric comparisons.
- Reading an input line does not split it for you. You get to split it to an array yourself. And the *split()* operator has different arguments than **awk**'s.
- The current input line is normally in `$_`, not `$0`. It generally does not have the newline stripped. ($0 is the name of the program executed.) See the *perlvar* manpage.
- $<*digit*> does not refer to fields—it refers to substrings matched by the last match pattern.
- The *print()* statement does not add field and record separators unless you set `$,` and `$\`. You can set `$OFS` and `$ORS` if you're using the English module.
- You must open your files before you print to them.
- The range operator is “..”, not comma. The comma operator works as in C.
- The match operator is “=~”, not “~”. ("~” is the one's complement operator, as in C.)
- The exponentiation operator is “**”, not “^”. “^” is the XOR operator, as in C. (You know, one could get the feeling that **awk** is basically incompatible with C.)
- The concatenation operator is “.”, not the null string. (Using the null string would render `/pat/ /pat/` unparsable, because the third slash would be interpreted as a division operator—the tokenizer is in fact slightly context sensitive for operators like “/”, “?”, and “>”. And in fact, “.” itself can be the beginning of a number.)
- The `next`, `exit`, and `continue` keywords work differently.
- The following variables work differently:

```
Awk       Perl
ARGC      $#ARGV or scalar @ARGV
ARGV[0]   $0
FILENAME  $ARGV
FNR       $. - something
FS        (whatever you like)
NF        $#Fld, or some such
NR        $.
OFMT      $#
OFS       $,
ORS       $\
RLENGTH   length($&)
RS        $/
RSTART    length($`)
SUBSEP    $;
```

- You cannot set `$RS` to a pattern, only a string.
- When in doubt, run the **awk** construct through **a2p** and see what it gives you.

### C Traps

Cerebral C programmers should take note of the following:

- Curly brackets are required on `if`'s and `while`'s.
- You must use `elsif` rather than `else if`.
- The `break` and `continue` keywords from C become in Perl `last` and `next`, respectively. Unlike in C, these do *NOT* work within a `do { } while` construct.
- There's no switch statement. (But it's easy to build one on the fly.)
- Variables begin with "$", "@" or "%" in Perl.
- `printf()` does not implement the "*" format for interpolating field widths, but it's trivial to use interpolation of double-quoted strings to achieve the same effect.
- Comments begin with "#", not "/*".
- You can't take the address of anything, although a similar operator in Perl is the backslash, which creates a reference.
- `ARGV` must be capitalized. `$ARGV[0]` is C's `argv[1]`, and `argv[0]` ends up in `$0`.
- System calls such as *link()*, *unlink()*, *rename()*, etc. return nonzero for success, not 0.
- Signal handlers deal with signal names, not numbers. Use `kill -l` to find their names on your system.

### Sed Traps

Seasoned **sed** programmers should take note of the following:

- Backreferences in substitutions use "$" rather than "\".
- The pattern matching metacharacters "(", ")", and "|" do not have backslashes in front.
- The range operator is `...`, rather than comma.

### Shell Traps

Sharp shell programmers should take note of the following:

- The backtick operator does variable interpolation without regard to the presence of single quotes in the command.
- The backtick operator does no translation of the return value, unlike **csh**.
- Shells (especially **csh**) do several levels of substitution on each command line. Perl does substitution in only certain constructs such as double quotes, backticks, angle brackets, and search patterns.
- Shells interpret scripts a little bit at a time. Perl compiles the entire program before executing it (except for `BEGIN` blocks, which execute at compile time).
- The arguments are available via `@ARGV`, not `$1`, `$2`, etc.
- The environment is not automatically made available as separate scalar variables.

### Perl Traps

Practicing Perl Programmers should take note of the following:

- Remember that many operations behave differently in a list context than they do in a scalar one. See the *perldata* manpage for details.
- Avoid barewords if you can, especially all lowercase ones. You can't tell by just looking at it whether a bareword is a function or a string. By using quotes on strings and parentheses on function calls, you won't ever get them confused.
- You cannot discern from mere inspection which builtins are unary operators (like *chop()* and *chdir()*) and which are list operators (like *print()* and *unlink()*). (User-defined subroutines can be **only** list operators, never unary ones.) See the *perlop* manpage.
- People have a hard time remembering that some functions default to `$_`, or `@ARGV`, or whatever, but that others which you might expect to do not.
- The <FH> construct is not the name of the filehandle, it is a readline operation on that handle. The data read is assigned to `$_` only if the file read is the sole condition in a while loop:

```
while (<FH>)      { }
while (defined($_ = <FH>)) { }..
<FH>;  # data discarded!
```

- Remember not to use "`=`" when you need "`=~`"; these two constructs are quite different:

```
$x =  /foo/;
$x =~ /foo/;
```

- The `do {}` construct isn't a real loop that you can use loop control on.
- Use `my()` for local variables whenever you can get away with it (but see the *perlform* manpage for where you can't). Using `local()` actually gives a local value to a global variable, which leaves you open to unforeseen side-effects of dynamic scoping.
- If you localize an exported variable in a module, its exported value will not change. The local name becomes an alias to a new value but the external name is still an alias for the original.

### Perl4 to Perl5 Traps

Practicing Perl4 Programmers should take note of the following Perl4-to-Perl5 specific traps.

They're crudely ordered according to the following list:

Discontinuance, Deprecation, and BugFix traps
: Anything that's been fixed as a perl4 bug, removed as a perl4 feature or deprecated as a perl4 feature with the intent to encourage usage of some other perl5 feature.

Parsing Traps
: Traps that appear to stem from the new parser.

Numerical Traps
: Traps having to do with numerical or mathematical operators.

General data type traps
: Traps involving perl standard data types.

Context Traps − scalar, list contexts
: Traps related to context within lists, scalar statements/declarations.

Precedence Traps
: Traps related to the precedence of parsing, evaluation, and execution of code.

General Regular Expression Traps using s///, etc.
: Traps related to the use of pattern matching.

Subroutine, Signal, Sorting Traps
: Traps related to the use of signals and signal handlers, general subroutines, and sorting, along with sorting subroutines.

OS Traps
: OS−specific traps.

DBM Traps
: Traps specific to the use of `dbmopen()`, and specific dbm implementations.

Unclassified Traps
: Everything else.

If you find an example of a conversion trap that is not listed here, please submit it to Bill Middleton <*wjm@best.com*> for inclusion. Also note that at least some of these can be caught with **−w**.

### Discontinuance, Deprecation, and BugFix traps

Anything that has been discontinued, deprecated, or fixed as a bug from perl4.

• Discontinuance
: Symbols starting with "_" are no longer forced into package main, except for `$_` itself (and `@_`, etc.).

```
package test;
$_legacy = 1;

package main;
print "\$_legacy is ",$_legacy,"\n";

# perl4 prints: $_legacy is 1
# perl5 prints: $_legacy is
```

• Deprecation
: Double-colon is now a valid package separator in a variable name. Thus these behave differently in perl4 vs. perl5, because the packages don't exist.

```
$a=1;$b=2;$c=3;$var=4;
print "$a::$b::$c ";
print "$var::abc::xyz\n";

# perl4 prints: 1::2::3 4::abc::xyz
# perl5 prints: 3
```

Given that `::` is now the preferred package delimiter, it is debatable whether this should be classed as a bug or not. (The older package delimiter, ' ,is used here)

```
$x = 10 ;
print "x=${'x}\n" ;

# perl4 prints: x=10
# perl5 prints: Can't find string terminator "'" anywhere before EOF
```

You can avoid this problem, and remain compatible with perl4, if you always explicitly include the package name:

```
$x = 10 ;
print "x=${main'x}\n" ;
```

Also see precedence traps, for parsing `$:`.

- BugFix
  The second and third arguments of `splice()` are now evaluated in scalar context (as the Camel says) rather than list context.

```
sub sub1{return(0,2) }          # return a 2-element list
sub sub2{ return(1,2,3)}        # return a 3-element list
@a1 = ("a","b","c","d","e");
@a2 = splice(@a1,&sub1,&sub2);
print join(' ',@a2),"\n";

# perl4 prints: a b
# perl5 prints: c d e
```

- Discontinuance
  You can't do a `goto` into a block that is optimized away. Darn.

```
goto marker1;

for(1){
marker1:
    print "Here I is!\n";
}

# perl4 prints: Here I is!
# perl5 dumps core (SEGV)
```

- Discontinuance
  It is no longer syntactically legal to use whitespace as the name of a variable, or as a delimiter for any kind of quote construct. Double darn.

```
$a = ("foo bar");
$b = q baz ;
print "a is $a, b is $b\n";

# perl4 prints: a is foo bar, b is baz
# perl5 errors: Bareword found where operator expected
```

- Discontinuance

  The archaic while/if BLOCK BLOCK syntax is no longer supported.

```
if { 1 } {
    print "True!";
}
else {
    print "False!";
}

# perl4 prints: True!
# perl5 errors: syntax error at test.pl line 1, near "if {"
```

- BugFix

  The `**` operator now binds more tightly than unary minus. It was documented to work this way before, but didn't.

```
print -4**2,"\n";

# perl4 prints: 16
# perl5 prints: -16
```

- Discontinuance

  The meaning of `foreach{}` has changed slightly when it is iterating over a list which is not an array. This used to assign the list to a temporary array, but no longer does so (for efficiency). This means that you'll now be iterating over the actual values, not over copies of the values. Modifications to the loop variable can change the original values.

```
@list = ('ab','abc','bcd','def');
foreach $var (grep(/ab/,@list)){
    $var = 1;
}
print (join(':',@list));

# perl4 prints: ab:abc:bcd:def
# perl5 prints: 1:1:bcd:def
```

  To retain Perl4 semantics you need to assign your list explicitly to a temporary array and then iterate over that. For example, you might need to change

```
foreach $var (grep(/ab/,@list)){
```

  to

```
foreach $var (@tmp = grep(/ab/,@list)){
```

  Otherwise changing `$var` will clobber the values of `@list`. (This most often happens when you use `$_` for the loop variable, and call subroutines in the loop that don't properly localize `$_`.)

• Discontinuance

split with no arguments now behaves like split ' ' (which doesn't return an initial null field if $_ starts with whitespace), it used to behave like split /\s+/ (which does).

```
$_ = ' hi mom';
print join(':', split);

# perl4 prints: :hi:mom
# perl5 prints: hi:mom
```

• BugFix

Perl 4 would ignore any text which was attached to an **–e** switch, always taking the code snippet from the following arg. Additionally, it would silently accept an **–e** switch without a following arg. Both of these behaviors have been fixed.

```
perl -e'print "attached to -e"' 'print "separate arg"'

# perl4 prints: separate arg
# perl5 prints: attached to -e

perl -e

# perl4 prints:
# perl5 dies: No code specified for -e.
```

• Discontinuance

In Perl 4 the return value of push was undocumented, but it was actually the last value being pushed onto the target list. In Perl 5 the return value of push is documented, but has changed, it is the number of elements in the resulting list.

```
@x = ('existing');
print push(@x, 'first new', 'second new');

# perl4 prints: second new
# perl5 prints: 3
```

• Discontinuance

In Perl 4 (and versions of Perl 5 before 5.004), '\r' characters in Perl code were silently allowed, although they could cause (mysterious!) failures in certain constructs, particularly here documents. Now, '\r' characters cause an immediate fatal error. (Note: In this example, the notation **\015** represents the incorrect line ending. Depending upon your text viewer, it will look different.)

```
print "foo";\015
print "bar";

# perl4     prints: foobar
# perl5.003 prints: foobar
# perl5.004 dies: Illegal character \015 (carriage return)
```

See the *perldiag* manpage for full details.

• Deprecation

Some error messages will be different.

- Discontinuance
  Some bugs may have been inadvertently removed. :−)

**Parsing Traps**

Perl4-to-Perl5 traps from having to do with parsing.

- Parsing
  Note the space between . and =

```
$string . = "more string";
print $string;

# perl4 prints: more string
# perl5 prints: syntax error at - line 1, near ". ="
```

- Parsing
  Better parsing in perl 5

```
sub foo {}
&foo
print("hello, world\n");

# perl4 prints: hello, world
# perl5 prints: syntax error
```

- Parsing
  “if it looks like a function, it is a function” rule.

```
print
  ($foo == 1) ? "is one\n" : "is zero\n";

  # perl4 prints: is zero
  # perl5 warns: "Useless use of a constant in void context" if using -w
```

- Parsing
  String interpolation of the `$#array` construct differs when braces are to used around the name.

```
@ = (1..3);
print "${#a}";

# perl4 prints: 2
# perl5 fails with syntax error

@ = (1..3);
print "$#{a}";

# perl4 prints: {a}
# perl5 prints: 2
```

**Numerical Traps**

Perl4-to-Perl5 traps having to do with numerical operators, operands, or output from same.

- Numerical

  Formatted output and significant digits

```
print 7.373504 - 0, "\n";
printf "%20.18f\n", 7.373504 - 0;

# Perl4 prints:
7.375039999999996141
7.37503999999999614

# Perl5 prints:
7.373504
7.37503999999999614
```

- Numerical

  This specific item has been deleted. It demonstrated how the auto-increment operator would not catch when a number went over the signed int limit. Fixed in version 5.003_04. But always be wary when using large integers. If in doubt:

```
use Math::BigInt;
```

- Numerical

  Assignment of return values from numeric equality tests does not work in perl5 when the test evaluates to false (0). Logical tests now return an null, instead of 0

```
$p = ($test == 1);
print $p,"\n";

# perl4 prints: 0
# perl5 prints:
```

  Also see the section on *General Regular Expression Traps using s///, etc.* for another example of this new feature...

### General data type traps

Perl4-to-Perl5 traps involving most data-types, and their usage within certain expressions and/or context.

- (Arrays)

  Negative array subscripts now count from the end of the array.

```
@a = (1, 2, 3, 4, 5);
print "The third element of the array is $a[3] also expressed as $a[-2] \n";

# perl4 prints: The third element of the array is 4 also expressed as
# perl5 prints: The third element of the array is 4 also expressed as 4
```

- (Arrays)

  Setting `$#array` lower now discards array elements, and makes them impossible to recover.

```
@a = (a,b,c,d,e);
print "Before: ",join('',@a);
$#a =1;
print ", After: ",join('',@a);
$#a =3;
print ", Recovered: ",join('',@a),"\n";
```

```
# perl4 prints: Before: abcde, After: ab, Recovered: abcd
# perl5 prints: Before: abcde, After: ab, Recovered: ab
```

- (Hashes)
  Hashes get defined before use

```
local($s,@a,%h);
die "scalar \$s defined" if defined($s);
die "array \@a defined" if defined(@a);
die "hash \%h defined" if defined(%h);

# perl4 prints:
# perl5 dies: hash %h defined
```

- (Globs)
  glob assignment from variable to variable will fail if the assigned variable is localized subsequent to the assignment

```
@a = ("This is Perl 4");
*b = *a;
local(@a);
print @b,"\n";

# perl4 prints: This is Perl 4
# perl5 prints:
```

- (Globs)
  Assigning `undef` to a glob has no effect in Perl 5. In Perl 4 it undefines the associated scalar (but may have other side effects including SEGVs).
- (Scalar String)
  Changes in unary negation (of strings) This change effects both the return value and what it does to *auto* (magic)increment.

```
$x = "aaa";
print ++$x," : ";
print -$x," : ";
print ++$x,"\n";

# perl4 prints: aab : -0 : 1
# perl5 prints: aab : -aab : aac
```

- (Constants)
  perl 4 lets you modify constants:

```
$foo = "x";
&mod($foo);
for ($x = 0; $x < 3; $x++) {
    &mod("a");
}
sub mod {
    print "before: $_[0]";
    $_[0] = "m";
    print "  after: $_[0]\n";
}
```

```
# perl4:
# before: x  after: m
# before: a  after: m
# before: m  after: m
# before: m  after: m

# Perl5:
# before: x  after: m
# Modification of a read-only value attempted at foo.pl line 12.
# before: a
```

- (Scalars)
  The behavior is slightly different for:

```
print "$x", defined $x

# perl 4: 1
# perl 5: <no output, $x is not called into existence>
```

- (Variable Suicide)
  Variable suicide behavior is more consistent under Perl 5. Perl5 exhibits the same behavior for hashes and scalars, that perl4 exhibits for only scalars.

```
$aGlobal{ "aKey" } = "global value";
print "MAIN:", $aGlobal{"aKey"}, "\n";
$GlobalLevel = 0;
&test( *aGlobal );

sub test {
    local( *theArgument ) = @_;
    local( %aNewLocal ); # perl 4 != 5.001l,m
    $aNewLocal{"aKey"} = "this should never appear";
    print "SUB: ", $theArgument{"aKey"}, "\n";
    $aNewLocal{"aKey"} = "level $GlobalLevel";   # what should print
    $GlobalLevel++;
    if( $GlobalLevel<4 ) {
        &test( *aNewLocal );
    }
}

# Perl4:
# MAIN:global value
# SUB: global value
# SUB: level 0
# SUB: level 1
# SUB: level 2

# Perl5:
# MAIN:global value
# SUB: global value
# SUB: this should never appear
# SUB: this should never appear
# SUB: this should never appear
```

**Context Traps – scalar, list contexts**

• (list context)

The elements of argument lists for formats are now evaluated in list context. This means you can interpolate list values now.

```
@fmt = ("foo","bar","baz");
format STDOUT=
@<<<<< @||||| @>>>>>
@fmt;
.
write;

# perl4 errors:  Please use commas to separate fields in file
# perl5 prints: foo     bar      baz
```

• (scalar context)

The `caller()` function now returns a false value in a scalar context if there is no caller. This lets library files determine if they're being required.

```
caller() ? (print "You rang?\n") : (print "Got a 0\n");

# perl4 errors: There is no caller
# perl5 prints: Got a 0
```

• (scalar context)

The comma operator in a scalar context is now guaranteed to give a scalar context to its arguments.

```
@y= ('a','b','c');
$x = (1, 2, @y);
print "x = $x\n";

# Perl4 prints:  x = c   # Thinks list context interpolates list
# Perl5 prints:  x = 3   # Knows scalar uses length of list
```

• (list, builtin)

`sprintf()` funkiness (array argument converted to scalar array count) This test could be added to t/op/sprintf.t

```
@z = ('%s%s', 'foo', 'bar');
$x = sprintf(@z);
if ($x eq 'foobar') {print "ok 2\n";} else {print "not ok 2 '$x'\n";}

# perl4 prints: ok 2
# perl5 prints: not ok 2
```

`printf()` works fine, though:

```
printf STDOUT (@z);
print "\n";

# perl4 prints: foobar
# perl5 prints: foobar
```

Probably a bug.

### Precedence Traps

Perl4-to-Perl5 traps involving precedence order.

Perl 4 has almost the same precedence rules as Perl 5 for the operators that they both have. Perl 4 however, seems to have had some inconsistencies that made the behavior differ from what was documented.

- Precedence

  LHS vs. RHS of any assignment operator. LHS is evaluated first in perl4, second in perl5; this can affect the relationship between side-effects in sub-expressions.

```
@arr = ( 'left', 'right' );
$a{shift @arr} = shift @arr;
print join( ' ', keys %a );

# perl4 prints: left
# perl5 prints: right
```

- Precedence

  These are now semantic errors because of precedence:

```
@list = (1,2,3,4,5);
%map = ("a",1,"b",2,"c",3,"d",4);
$n = shift @list + 2;   # first item in list plus 2
print "n is $n, ";
$m = keys %map + 2;     # number of items in hash plus 2
print "m is $m\n";

# perl4 prints: n is 3, m is 6
# perl5 errors and fails to compile
```

- Precedence

  The precedence of assignment operators is now the same as the precedence of assignment. Perl 4 mistakenly gave them the precedence of the associated operator. So you now must parenthesize them in expressions like

```
/foo/ ? ($a += 2) : ($a -= 2);
```

  Otherwise

```
/foo/ ? $a += 2 : $a -= 2
```

  would be erroneously parsed as

```
(/foo/ ? $a += 2 : $a) -= 2;
```

  On the other hand,

```
$a += /foo/ ? 1 : 2;
```

  now works as a C programmer would expect.

- Precedence

```
open FOO || die;
```

  is now incorrect. You need parentheses around the filehandle. Otherwise, perl5 leaves the statement as its default precedence:

```
open(FOO || die);

# perl4 opens or dies
# perl5 errors: Precedence problem: open FOO should be open(FOO)
```

- Precedence

  perl4 gives the special variable, `$:` precedence, where perl5 treats `$::` as main `package`

```
$a = "x"; print "$::a";

# perl 4 prints: -:a
# perl 5 prints: x
```

- Precedence

  perl4 had buggy precedence for the file test operators vis-a-vis the assignment operators. Thus, although the precedence table for perl4 leads one to believe `-e $foo .= "q"` should parse as `((-e $foo) .= "q")`, it actually parses as `(-e ($foo .= "q"))`. In perl5, the precedence is as documented.

```
-e $foo .= "q"

# perl4 prints: no output
# perl5 prints: Can't modify -e in concatenation
```

- Precedence

  In perl4, *keys()*, *each()* and *values()* were special high-precedence operators that operated on a single hash, but in perl5, they are regular named unary operators. As documented, named unary operators have lower precedence than the arithmetic and concatenation operators `+ - .`, but the perl4 variants of these operators actually bind tighter than `+ - .`. Thus, for:

```
%foo = 1..10;
print keys %foo - 1

# perl4 prints: 4
# perl5 prints: Type of arg 1 to keys must be hash (not subtraction)
```

  The perl4 behavior was probably more useful, if less consistent.

**General Regular Expression Traps using s///, etc.**

All types of RE traps.

- Regular Expression

  `s'$lhs'$rhs'` now does no interpolation on either side. It used to interpolate `$lhs` but not `$rhs`. (And still does not match a literal '$' in string)

```
$a=1;$b=2;
$string = '1 2 $a $b';
$string =~ s'$a'$b';
print $string,"\n";

# perl4 prints: $b 2 $a $b
# perl5 prints: 1 2 $a $b
```

- Regular Expression

  `m//g` now attaches its state to the searched string rather than the regular expression. (Once the scope of a block is left for the sub, the state of the searched string is lost)

```
$_ = "ababab";
while(m/ab/g){
    &doit("blah");
}
sub doit{local($_) = shift; print "Got $_ "}

# perl4 prints: blah blah blah
# perl5 prints: infinite loop blah...
```

- Regular Expression

  Currently, if you use the `m//o` qualifier on a regular expression within an anonymous sub, *all* closures generated from that anonymous sub will use the regular expression as it was compiled when it was used the very first time in any such closure. For instance, if you say

```
sub build_match {
    my($left,$right) = @_;
    return sub { $_[0] =~ /$left stuff $right/o; };
}
```

  *build_match()* will always return a sub which matches the contents of `$left` and `$right` as they were the *first* time that *build_match()* was called, not as they are in the current call.

  This is probably a bug, and may change in future versions of Perl.

- Regular Expression

  If no parentheses are used in a match, Perl4 sets `$+` to the whole match, just like `$&`. Perl5 does not.

```
"abcdef" =~ /b.*e/;
print "\$+ = $+\n";

# perl4 prints: bcde
# perl5 prints:
```

- Regular Expression

  substitution now returns the null string if it fails

```
$string = "test";
$value = ($string =~ s/foo//);
print $value, "\n";

# perl4 prints: 0
# perl5 prints:
```

  Also see the section on *Numerical Traps* for another example of this new feature.

- Regular Expression

  ``s`lhs`rhs` `` (using backticks) is now a normal substitution, with no backtick expansion

```
$string = "";
$string =~ s`^`hostname`;
print $string, "\n";
```

```
# perl4 prints: <the local hostname>
# perl5 prints: hostname
```

• Regular Expression

Stricter parsing of variables used in regular expressions

```
s/^([^$grpc]*$grpc[$opt$plus$rep]?)//o;

# perl4: compiles w/o error
# perl5: with Scalar found where operator expected ..., near "$opt$plus"
```

an added component of this example, apparently from the same script, is the actual value of the s'd string after the substitution. `[$opt]` is a character class in perl4 and an array subscript in perl5

```
$grpc = 'a';
$opt  = 'r';
$_ = 'bar';
s/^([^$grpc]*$grpc[$opt]?)/foo/;
print ;

# perl4 prints: foo
# perl5 prints: foobar
```

• Regular Expression

Under perl5, `m?x?` matches only once, like `?x?`. Under perl4, it matched repeatedly, like `/x/` or `m!x!`.

```
$test = "once";
sub match { $test =~ m?once?; }
&match();
if( &match() ) {
    # m?x? matches more then once
    print "perl4\n";
} else {
    # m?x? matches only once
    print "perl5\n";
}

# perl4 prints: perl4
# perl5 prints: perl5
```

**Subroutine, Signal, Sorting Traps**

The general group of Perl4-to-Perl5 traps having to do with Signals, Sorting, and their related subroutines, as well as general subroutine traps. Includes some OS−Specific traps.

• (Signals)

Barewords that used to look like strings to Perl will now look like subroutine calls if a subroutine by that name is defined before the compiler sees them.

```
sub SeeYa { warn"Hasta la vista, baby!" }
$SIG{'TERM'} = SeeYa;
print "SIGTERM is now $SIG{'TERM'}\n";
```

```
# perl4 prints: SIGTERM is main'SeeYa
# perl5 prints: SIGTERM is now main::1
```

Use **-w** to catch this one

- (Sort Subroutine)
  reverse is no longer allowed as the name of a sort subroutine.

```
sub reverse{ print "yup "; $a <=> $b }
print sort reverse a,b,c;

# perl4 prints: yup yup yup yup abc
# perl5 prints: abc
```

- warn() won't let you specify a filehandle.
  Although it _always_ printed to STDERR, *warn()* would let you specify a filehandle in perl4. With perl5 it does not.

```
warn STDERR "Foo!";

# perl4 prints: Foo!
# perl5 prints: String found where operator expected
```

**OS Traps**

- (SysV)
  Under HPUX, and some other SysV OSes, one had to reset any signal handler, within the signal handler function, each time a signal was handled with perl4. With perl5, the reset is now done correctly. Any code relying on the handler _not_ being reset will have to be reworked.

  Since version 5.002, Perl uses *sigaction()* under SysV.

```
sub gotit {
    print "Got @_... ";
}
$SIG{'INT'} = 'gotit';

$| = 1;
$pid = fork;
if ($pid) {
    kill('INT', $pid);
    sleep(1);
    kill('INT', $pid);
} else {
    while (1) {sleep(10);}
}

# perl4 (HPUX) prints: Got INT...
# perl5 (HPUX) prints: Got INT... Got INT...
```

- (SysV)
  Under SysV OSes, `seek()` on a file opened to append >> now does the right thing w.r.t. the *fopen()* manpage. e.g., – When a file is opened for append, it is impossible to overwrite information already in the file.

```
open(TEST,">>seek.test");
$start = tell TEST ;
foreach(1 .. 9){
    print TEST "$_ ";
}
$end = tell TEST ;
seek(TEST,$start,0);
print TEST "18 characters here";

# perl4 (solaris) seek.test has: 18 characters here
# perl5 (solaris) seek.test has: 1 2 3 4 5 6 7 8 9 18 characters here
```

### Interpolation Traps

Perl4-to-Perl5 traps having to do with how things get interpolated within certain expressions, statements, contexts, or whatever.

- Interpolation

  @ now always interpolates an array in double-quotish strings.

```
print "To: someone@somewhere.com\n";

# perl4 prints: To:someone@somewhere.com
# perl5 errors : In string, @somewhere now must be written as \@somewhere
```

- Interpolation

  Double-quoted strings may no longer end with an unescaped $ or @.

```
$foo = "foo$";
$bar = "bar@";
print "foo is $foo, bar is $bar\n";

# perl4 prints: foo is foo$, bar is bar@
# perl5 errors: Final $ should be \$ or $name
```

  Note: perl5 DOES NOT error on the terminating @ in `$bar`

- Interpolation

  Perl now sometimes evaluates arbitrary expressions inside braces that occur within double quotes (usually when the opening brace is preceded by `$` or `@`).

```
@www = "buz";
$foo = "foo";
$bar = "bar";
sub foo { return "bar" };
print "|@{w.w.w}|${main'foo}|";

# perl4 prints: |@{w.w.w}|foo|
# perl5 prints: |buz|bar|
```

  Note that you can `use strict;` to ward off such trappiness under perl5.

- Interpolation

  The construct "this is $$x" used to interpolate the pid at that point, but now apparently tries to dereference `$x`. `$$` by itself still works fine, however.

```
print "this is $$x\n";
```

```
# perl4 prints: this is XXXx   (XXX is the current pid)
# perl5 prints: this is
```

- Interpolation

  Creation of hashes on the fly with `eval "EXPR"` now requires either both `$`'s to be protected in the specification of the hash name, or both curlies to be protected. If both curlies are protected, the result will be compatible with perl4 and perl5. This is a very common practice, and should be changed to use the block form of `eval{}` if possible.

```
$hashname = "foobar";
$key = "baz";
$value = 1234;
eval "\$$hashname{'$key'} = q|$value|";
(defined($foobar{'baz'})) ?  (print "Yup") : (print "Nope");

# perl4 prints: Yup
# perl5 prints: Nope
```

  Changing

```
eval "\$$hashname{'$key'} = q|$value|";
```

  to

```
eval "\$\$hashname{'$key'} = q|$value|";
```

  causes the following result:

```
# perl4 prints: Nope
# perl5 prints: Yup
```

  or, changing to

```
eval "\$$hashname\{'$key'\} = q|$value|";
```

  causes the following result:

```
# perl4 prints: Yup
# perl5 prints: Yup
# and is compatible for both versions
```

- Interpolation

  perl4 programs which unconsciously rely on the bugs in earlier perl versions.

```
perl -e '$bar=q/not/; print "This is $foo{$bar} perl5"'

# perl4 prints: This is not perl5
# perl5 prints: This is perl5
```

- Interpolation

  You also have to be careful about array references.

```
print "$foo{"

perl 4 prints: {
perl 5 prints: syntax error
```

• Interpolation

Similarly, watch out for:

```
$foo = "array";
print "\$$foo{bar}\n";

# perl4 prints: $array{bar}
# perl5 prints: $
```

Perl 5 is looking for `$array{bar}` which doesn't exist, but perl 4 is happy just to expand `$foo` to "array" by itself. Watch out for this especially in `eval`'s.

• Interpolation

`qq()` string passed to `eval`

```
eval qq(
    foreach \$y (keys %\$x\) {
        \$count++;
    }
);

# perl4 runs this ok
# perl5 prints: Can't find string terminator ")"
```

**DBM Traps**

General DBM traps.

• DBM

Existing dbm databases created under perl4 (or any other dbm/ndbm tool) may cause the same script, run under perl5, to fail. The build of perl5 must have been linked with the same dbm/ndbm as the default for `dbmopen()` to function properly without `tie`'ing to an extension dbm implementation.

```
dbmopen (%dbm, "file", undef);
print "ok\n";

# perl4 prints: ok
# perl5 prints: ok (IFF linked with -ldbm or -lndbm)
```

• DBM

Existing dbm databases created under perl4 (or any other dbm/ndbm tool) may cause the same script, run under perl5, to fail. The error generated when exceeding the limit on the key/value size will cause perl5 to exit immediately.

```
dbmopen(DB, "testdb",0600) || die "couldn't open db! $!";
$DB{'trap'} = "x" x 1024;  # value too large for most dbm/ndbm
print "YUP\n";

# perl4 prints:
dbm store returned -1, errno 28, key "trap" at - line 3.
YUP

# perl5 prints:
dbm store returned -1, errno 28, key "trap" at - line 3.
```

**Unclassified Traps**

Everything else.

- `require`/`do` trap using returned value

  If the file doit.pl has:

```
sub foo {
    $rc = do "./do.pl";
    return 8;
}
print &foo, "\n";
```

  And the do.pl file has the following single line:

```
return 3;
```

  Running doit.pl gives the following:

```
# perl 4 prints: 3 (aborts the subroutine early)
# perl 5 prints: 8
```

  Same behavior if you replace `do` with `require`.

- `split` on empty string with LIMIT specified

```
    $string = '';
@list = split(/foo/, $string, 2)
```

  Perl4 returns a one element list containing the empty string but Perl5 returns an empty list.

As always, if any of these are ever officially declared as bugs, they'll be fixed and removed.

## NAME

perlport − Writing portable Perl

## DESCRIPTION

Perl runs on a variety of operating systems. While most of them share a lot in common, they also have their own very particular and unique features.

This document is meant to help you to find out what constitutes portable perl code, so that once you have made your decision to write portably, you know where the lines are drawn, and you can stay within them.

There is a tradeoff between taking full advantage of **a** particular type of computer, and taking advantage of a full **range** of them. Naturally, as you make your range bigger (and thus more diverse), the common denominators drop, and you are left with fewer areas of common ground in which you can operate to accomplish a particular task. Thus, when you begin attacking a problem, it is important to consider which part of the tradeoff curve you want to operate under. Specifically, whether it is important to you that the task that you are coding needs the full generality of being portable, or if it is sufficient to just get the job done. This is the hardest choice to be made. The rest is easy, because Perl provides lots of choices, whichever way you want to approach your problem.

Looking at it another way, writing portable code is usually about willfully limiting your available choices. Naturally, it takes discipline to do that.

Be aware of two important points:

Not all Perl programs have to be portable

There is no reason why you should not use Perl as a language to glue Unix tools together, or to prototype a Macintosh application, or to manage the Windows registry. If it makes no sense to aim for portability for one reason or another in a given program, then don't bother.

The vast majority of Perl **is** portable

Don't be fooled into thinking that it is hard to create portable Perl code. It isn't. Perl tries its level-best to bridge the gaps between what's available on different platforms, and all the means available to use those features. Thus almost all Perl code runs on any machine without modification. But there *are* some significant issues in writing portable code, and this document is entirely about those issues.

Here's the general rule: When you approach a task that is commonly done using a whole range of platforms, think in terms of writing portable code. That way, you don't sacrifice much by way of the implementation choices you can avail yourself of, and at the same time you can give your users lots of platform choices. On the other hand, when you have to take advantage of some unique feature of a particular platform, as is often the case with systems programming (whether for Unix, Windows, Mac OS, VMS, etc.), consider writing platform-specific code.

When the code will run on only two or three operating systems, then you may only need to consider the differences of those particular systems. The important thing is to decide where the code will run, and to be deliberate in your decision.

This information should not be considered complete; it includes possibly transient information about idiosyncracies of some of the ports, almost all of which are in a state of constant evolution. Thus this material should be considered a perpetual work in progress (<IMG SRC="yellow_sign.gif'' ALT="Under Construction">).

## ISSUES

### Newlines

In most operating systems, lines in files are separated with newlines. Just what is used as a newline may vary from OS to OS. Unix traditionally uses `\012`, one kind of Windows I/O uses `\015\012`, and Mac OS uses `\015`.

Perl uses `\n` to represent the ''logical'' newline, where what is logical may depend on the platform in use. In MacPerl, `\n` always means `\015`. In DOSish perls, `\n` usually means `\012`, but when accessing a file in ''text'' mode, STDIO translates it to (or from) `\015\012`.

Due to the ''text'' mode translation, DOSish perls have limitations of using `seek` and `tell` when a file is being accessed in "text" mode. Specifically, if you stick to `seek`–ing to locations you got from `tell` (and no others), you are usually free to use `seek` and `tell` even in "text" mode. In general, using `seek` or `tell` or other file operations that count bytes instead of characters, without considering the length of `\n`, may be non-portable. If you use `binmode` on a file, however, you can usually use `seek` and `tell` with arbitrary values quite safely.

A common misconception in socket programming is that `\n` eq `\012` everywhere. When using protocols, such as common Internet protocols, `\012` and `\015` are called for specifically, and the values of the logical `\n` and `\r` (carriage return) are not reliable.

```
print SOCKET "Hi there, client!\r\n";      # WRONG
print SOCKET "Hi there, client!\015\012";  # RIGHT
```

[NOTE: this does not necessarily apply to communications that are filtered by another program or module before sending to the socket; the the most popular EBCDIC webserver, for instance, accepts `\r\n`, which translates those characters, along with all other characters in text streams, from EBCDIC to ASCII.]

However, `\015\012` (or `\cM\cJ`, or `\x0D\x0A`) can be tedious and unsightly, as well as confusing to those maintaining the code. As such, the `Socket` module supplies the Right Thing for those who want it.

```
use Socket qw(:DEFAULT :crlf);
print SOCKET "Hi there, client!$CRLF"      # RIGHT
```

When reading *from* a socket, remember that the default input record separator (`$/`) is `\n`, but code like this should recognize `$/` as `\012` or `\015\012`:

```
while (<SOCKET>) {
    # ...
}
```

Better:

```
use Socket qw(:DEFAULT :crlf);
local($/) = LF;      # not needed if $/ is already \012

while (<SOCKET>) {
    s/$CR?$LF/\n/;   # not sure if socket uses LF or CRLF, OK
#   s/\015?\012/\n/; # same thing
}
```

And this example is actually better than the previous one even for Unix platforms, because now any `\015`'s (`\cM`'s) are stripped out (and there was much rejoicing).

### File Paths

Most platforms these days structure files in a hierarchical fashion. So, it is reasonably safe to assume that any platform supports the notion of a "path" to uniquely identify a file on the system. Just how that path is actually written, differs.

While they are similar, file path specifications differ between Unix, Windows, Mac OS, OS/2, VMS and probably others. Unix, for example, is one of the few OSes that has the idea of a root directory. Mac OS uses `:` as a path separator instead of `/`. VMS, Windows, and OS/2 can work similarly to Unix with `/` as path separator, or in their own idiosyncratic ways.

As with the newline problem above, there are modules that can help. The `File::Spec` modules provide methods to do the Right Thing on whatever platform happens to be running the program.

```
use File::Spec;
chdir(File::Spec->updir());        # go up one directory
$file = File::Spec->catfile(
    File::Spec->curdir(), 'temp', 'file.txt'
);
# on Unix and Win32, './temp/file.txt'
# on Mac OS, ':temp:file.txt'
```

File::Spec is available in the standard distribution, as of version 5.004_05.

In general, production code should not have file paths hardcoded; making them user supplied or from a configuration file is better, keeping in mind that file path syntax varies on different machines.

This is especially noticeable in scripts like Makefiles and test suites, which often assume `/` as a path separator for subdirectories.

Also of use is `File::Basename`, from the standard distribution, which splits a pathname into pieces (base filename, full path to directory, and file suffix).

Remember not to count on the existence of system-specific files, like */etc/resolv.conf*. If code does need to rely on such a file, include a description of the file and its format in the code's documentation, and make it easy for the user to override the default location of the file.

### System Interaction

Not all platforms provide for the notion of a command line, necessarily. These are usually platforms that rely on a Graphical User Interface (GUI) for user interaction. So a program requiring command lines might not work everywhere. But this is probably for the user of the program to deal with.

Some platforms can't delete or rename files that are being held open by the system. Remember to `close` files when you are done with them. Don't `unlink` or `rename` an open file. Don't `tie` to or `open` a file that is already tied to or opened; `untie` or `close` first.

Don't count on a specific environment variable existing in `%ENV`. Don't even count on `%ENV` entries being case-sensitive, or even case-preserving.

Don't count on signals in portable programs.

Don't count on filename globbing. Use `opendir`, `readdir`, and `closedir` instead.

### Interprocess Communication (IPC)

In general, don't directly access the system in code that is meant to be portable. That means, no: `system`, `exec`, `fork`, `pipe`, ``` `` ```, `qx//`, `open` with a `|`, or any of the other things that makes being a Unix perl hacker worth being.

Commands that launch external processes are generally supported on most platforms (though many of them do not support any type of forking), but the problem with using them arises from what you invoke with them. External tools are often named differently on different platforms, often not available in the same location, often accept different arguments, often behave differently, and often represent their results in a platform-dependent way. Thus you should seldom depend on them to produce consistent results.

One especially common bit of Perl code is opening a pipe to sendmail:

```
open(MAIL, '|/usr/lib/sendmail -t') or die $!;
```

This is fine for systems programming when sendmail is known to be available. But it is not fine for many non-Unix systems, and even some Unix systems that may not have sendmail installed. If a portable solution is needed, see the `Mail::Send` and `Mail::Mailer` modules in the `MailTools` distribution. `Mail::Mailer` provides several mailing methods, including mail, sendmail, and direct SMTP (via `Net::SMTP`) if a mail transfer agent is not available.

The rule of thumb for portable code is: Do it all in portable Perl, or use a module that may internally implement it with platform-specific code, but expose a common interface. By portable Perl, we mean code that avoids the constructs described in this document as being non-portable.

### External Subroutines (XS)

XS code, in general, can be made to work with any platform; but dependent libraries, header files, etc., might not be readily available or portable, or the XS code itself might be platform-specific, just as Perl code might be. If the libraries and headers are portable, then it is normally reasonable to make sure the XS code is portable, too.

There is a different kind of portability issue with writing XS code: availability of a C compiler on the end-user's system. C brings with it its own portability issues, and writing XS code will expose you to some of those. Writing purely in perl is a comparatively easier way to achieve portability.

### Standard Modules

In general, the standard modules work across platforms. Notable exceptions are `CPAN.pm` (which currently makes connections to external programs that may not be available), platform-specific modules (like `ExtUtils::MM_VMS`), and DBM modules.

There is no one DBM module that is available on all platforms. `SDBM_File` and the others are generally available on all Unix and DOSish ports, but not in MacPerl, where `NBDM_File` and `DB_File` are available.

The good news is that at least some DBM module should be available, and `AnyDBM_File` will use whichever module it can find. Of course, then the code needs to be fairly strict, dropping to the lowest common denominator (e.g., not exceeding 1K for each record).

### Time and Date

The system's notion of time of day and calendar date is controlled in widely different ways. Don't assume the timezone is stored in `$ENV{TZ}`, and even if it is, don't assume that you can control the timezone through that variable.

Don't assume that the epoch starts at January 1, 1970, because that is OS−specific. Better to store a date in an unambiguous representation. A text representation (like `1 Jan 1970`) can be easily converted into an OS−specific value using a module like `Date::Parse`. An array of values, such as those returned by `localtime`, can be converted to an OS−specific representation using `Time::Local`.

### System Resources

If your code is destined for systems with severely constrained (or missing!) virtual memory systems then you want to be especially mindful of avoiding wasteful constructs such as:

```
# NOTE: this is no longer "bad" in perl5.005
for (0..10000000) {}                        # bad
for (my $x = 0; $x <= 10000000; ++$x) {}    # good

@lines = <VERY_LARGE_FILE>;                 # bad

while (<FILE>) {$file .= $_}                # sometimes bad
$file = join '', <FILE>;                    # better
```

The last two may appear unintuitive to most people. The first of those two constructs repeatedly grows a string, while the second allocates a large chunk of memory in one go. On some systems, the latter is more efficient that the former.

### Security

Most Unix platforms provide basic levels of security that is usually felt at the file-system level. Other platforms usually don't (unfortunately). Thus the notion of User-ID, or "home" directory, or even the state of being logged-in may be unrecognizable on may platforms. If you write programs that are security conscious, it is usually best to know what type of system you will be operating under, and write code explicitly for that platform (or class of platforms).

### Style

For those times when it is necessary to have platform-specific code, consider keeping the platform-specific code in one place, making porting to other platforms easier. Use the `Config` module and the special variable `$^O` to differentiate platforms, as described in the section on *PLATFORMS*.

## CPAN TESTERS

Module uploaded to CPAN are tested by a variety of volunteers on different platforms. These CPAN testers are notified by e-mail of each new upload, and reply to the list with PASS, FAIL, NA (not applicable to this platform), or ???? (unknown), along with any relevant notations.

The purpose of the testing is twofold: one, to help developers fix any problems in their code; two, to provide users with information about whether or not a given module works on a given platform.

Mailing list: cpan-testers@perl.org

Testing results: `http://www.connect.net/gbarr/cpan-test/`

## PLATFORMS

As of version 5.002, Perl is built with a `$^O` variable that indicates the operating system it was built on. This was implemented to help speed up code that would otherwise have to `use Config;` and use the value of `$Config{'osname'}`. Of course, to get detailed information about the system, looking into `%Config` is certainly recommended.

### Unix

Perl works on a bewildering variety of Unix and Unix-like platforms (see e.g. most of the files in the *hints/* directory in the source code kit). On most of these systems, the value of `$^O` (hence `$Config{'osname'}`, too) is determined by lowercasing and stripping punctuation from the first field of the string returned by typing

```
% uname -a
```

(or a similar command) at the shell prompt. Here, for example, are a few of the more popular Unix flavors:

```
uname         $^O
--------------------
AIX           aix
FreeBSD       freebsd
Linux         linux
HP-UX         hpux
OSF1          dec_osf
SunOS         solaris
SunOS4        sunos
```

### DOS and Derivatives

Perl has long been ported to PC style microcomputers running under systems like PC-DOS, MS-DOS, OS/2, and most Windows platforms you can bring yourself to mention (except for Windows CE, if you count that). Users familiar with *COMMAND.COM* and/or *CMD.EXE* style shells should be aware that each of these file specifications may have subtle differences:

```
$filespec0 = "c:/foo/bar/file.txt";
$filespec1 = "c:\\foo\\bar\\file.txt";
$filespec2 = 'c:\foo\bar\file.txt';
$filespec3 = 'c:\\foo\\bar\\file.txt';
```

System calls accept either `/` or `\` as the path separator. However, many command-line utilities of DOS vintage treat `/` as the option prefix, so they may get confused by filenames containing `/`. Aside from calling any external programs, `/` will work just fine, and probably better, as it is more consistent with popular usage, and avoids the problem of remembering what to backwhack and what not to.

The DOS FAT file system can only accomodate "8.3" style filenames. Under the "case insensitive, but case preserving" HPFS (OS/2) and NTFS (NT) file systems you may have to be careful about case returned with functions like `readdir` or used with functions like `open` or `opendir`.

DOS also treats several filenames as special, such as AUX, PRN, NUL, CON, COM1, LPT1, LPT2 etc. Unfortunately these filenames won't even work if you include an explicit directory prefix, in some cases. It is best to avoid such filenames, if you want your code to be portable to DOS and its derivatives.

Users of these operating systems may also wish to make use of scripts such as *pl2bat.bat* or *pl2cmd* as appropriate to put wrappers around your scripts.

Newline (`\n`) is translated as `\015\012` by STDIO when reading from and writing to files. `binmode(FILEHANDLE)` will keep `\n` translated as `\012` for that filehandle. Since it is a noop on other systems, `binmode` should be used for cross-platform code that deals with binary data.

The `$^O` variable and the `$Config{'archname'}` values for various DOSish perls are as follows:

```
OS            $^O        $Config{'archname'}
--------------------------------------------
MS-DOS        dos
PC-DOS        dos
OS/2          os2
Windows 95    MSWin32    MSWin32-x86
Windows NT    MSWin32    MSWin32-x86
Windows NT    MSWin32    MSWin32-alpha
Windows NT    MSWin32    MSWin32-ppc
```

Also see:

The djgpp environment for DOS, `http://www.delorie.com/djgpp/`

The EMX environment for DOS, OS/2, etc. `emx@iaehv.nl`,
`http://www.juge.com/bbs/Hobb.19.html`

Build instructions for Win32, the *perlwin32* manpage.

The ActiveState Pages, `http://www.activestate.com/`

### MacPerl

Any module requiring XS compilation is right out for most people, because MacPerl is built using non-free (and non-cheap!) compilers. Some XS modules that can work with MacPerl are built and distributed in binary form on CPAN. See *MacPerl: Power and Ease* for more details.

Directories are specified as:

```
volume:folder:file              for absolute pathnames
volume:folder:                  for absolute pathnames
:folder:file                    for relative pathnames
:folder:                        for relative pathnames
:file                           for relative pathnames
file                            for relative pathnames
```

Files in a directory are stored in alphabetical order. Filenames are limited to 31 characters, and may include any character except `:`, which is reserved as a path separator.

Instead of `flock`, see `FSpSetFLock` and `FSpRstFLock` in `Mac::Files`.

In the MacPerl application, you can't run a program from the command line; programs that expect `@ARGV` to be populated can be edited with something like the following, which brings up a dialog box asking for the command line arguments.

```
if (!@ARGV) {
    @ARGV = split /\s+/, MacPerl::Ask('Arguments?');
}
```

A MacPerl script saved as a droplet will populate `@ARGV` with the full pathnames of the files dropped onto the script.

Mac users can use programs on a kind of command line under MPW (Macintosh Programmer's Workshop, a free development environment from Apple). MacPerl was first introduced as an MPW tool, and MPW can be used like a shell:

```
perl myscript.plx some arguments
```

ToolServer is another app from Apple that provides access to MPW tools from MPW and the MacPerl app, which allows MacPerl program to use `system`, backticks, and piped `open`.

"Mac OS" is the proper name for the operating system, but the value in `$^O` is "MacOS". To determine architecture, version, or whether the application or MPW tool version is running, check:

```
$is_app    = $MacPerl::Version =~ /App/;
$is_tool   = $MacPerl::Version =~ /MPW/;
($version) = $MacPerl::Version =~ /^(\S+)/;
$is_ppc    = $MacPerl::Architecture eq 'MacPPC';
$is_68k    = $MacPerl::Architecture eq 'Mac68K';
```

Also see:

The MacPerl Pages, `http://www.ptf.com/macperl/`.

The MacPerl mailing list, `mac-perl-request@iis.ee.ethz.ch`.

### VMS

Perl on VMS is discussed in *vms/perlvms.pod* in the perl distribution. Note that perl on VMS can accept either VMS or Unix style file specifications as in either of the following:

```
$ perl -ne "print if /perl_setup/i" SYS$LOGIN:LOGIN.COM
$ perl -ne "print if /perl_setup/i" /sys$login/login.com
```

but not a mixture of both as in:

```
$ perl -ne "print if /perl_setup/i" sys$login:/login.com
Can't open sys$login:/login.com: file specification syntax error
```

Interacting with Perl from the Digital Command Language (DCL) shell often requires a different set of quotation marks than Unix shells do. For example:

```
$ perl -e "print ""Hello, world.\n"""
Hello, world.
```

There are a number of ways to wrap your perl scripts in DCL .COM files if you are so inclined. For example:

```
$ write sys$output "Hello from DCL!"
$ if p1 .eqs. ""
$ then perl -x 'f$environment("PROCEDURE")
$ else perl -x - 'p1 'p2 'p3 'p4 'p5 'p6 'p7 'p8
$ deck/dollars="__END__"
#!/usr/bin/perl

print "Hello from Perl!\n";

__END__
$ endif
```

Do take care with `$ ASSIGN/nolog/user SYS$COMMAND: SYS$INPUT` if your perl-in-DCL script expects to do things like `$read = <STDIN>;`.

Filenames are in the format "name.extension;version". The maximum length for filenames is 39 characters, and the maximum length for extensions is also 39 characters. Version is a number from 1 to 32767. Valid characters are `/[A-Z0-9$_-]/`.

VMS' RMS filesystem is case insensitive and does not preserve case. `readdir` returns lowercased filenames, but specifying a file for opening remains case insensitive. Files without extensions have a trailing period on them, so doing a `readdir` with a file named *A.;5* will return *a.* (though that file could be opened with `open(FH, 'A')`.

RMS has an eight level limit on directory depths from any rooted logical (allowing 16 levels overall). Hence `PERL_ROOT:[LIB.2.3.4.5.6.7.8]` is a valid directory specification but `PERL_ROOT:[LIB.2.3.4.5.6.7.8.9]` is not. *Makefile.PL* authors might have to take this into account, but at least they can refer to the former as `/PERL_ROOT/lib/2/3/4/5/6/7/8/`.

The `VMS::Filespec` module, which gets installed as part of the build process on VMS, is a pure Perl module that can easily be installed on non-VMS platforms and can be helpful for conversions to and from RMS native formats.

What `\n` represents depends on the type of file that is open. It could be `\015`, `\012`, `\015\012`, or nothing. Reading from a file translates newlines to `\012`, unless `binmode` was executed on that handle, just like DOSish perls.

TCP/IP stacks are optional on VMS, so socket routines might not be implemented. UDP sockets may not be supported.

The value of `$^O` on OpenVMS is "VMS". To determine the architecture that you are running on without resorting to loading all of `%Config` you can examine the content of the `@INC` array like so:

```
if (grep(/VMS_AXP/, @INC)) {
    print "I'm on Alpha!\n";
} elsif (grep(/VMS_VAX/, @INC)) {
    print "I'm on VAX!\n";
} else {
    print "I'm not so sure about where $^O is...\n";
}
```

Also see:

the *perlvms.pod* manpage

vmsperl list, `vmsperl-request@newman.upenn.edu`
Put words `SUBSCRIBE VMSPERL` in message body.

vmsperl on the web, `http://www.sidhe.org/vmsperl/index.html`

**EBCDIC Platforms**

Recent versions of Perl have been ported to platforms such as OS/400 on AS/400 minicomputers as well as OS/390 for IBM Mainframes. Such computers use EBCDIC character sets internally (usually Character Code Set ID 00819 for OS/400 and IBM−1047 for OS/390). Note that on the mainframe perl currently works under the "Unix system services for OS/390" (formerly known as OpenEdition).

As of R2.5 of USS for OS/390 that Unix sub-system did not support the `#!` shebang trick for script invocation. Hence, on OS/390 perl scripts can executed with a header similar to the following simple script:

```
: # use perl
    eval 'exec /usr/local/bin/perl -S $0 ${1+"$@"}'
        if 0;
#!/usr/local/bin/perl     # just a comment really

print "Hello from perl!\n";
```

On these platforms, bear in mind that the EBCDIC character set may have an effect on what happens with perl functions such as `chr`, `pack`, `print`, `printf`, `ord`, `sort`, `sprintf`, `unpack`; as well as bit-fiddling with ASCII constants using operators like `^`, `&` and `|`; not to mention dealing with socket interfaces to ASCII computers (see the section on *NEWLINES*).

Fortunately, most web servers for the mainframe will correctly translate the `\n` in the following statement to its ASCII equivalent (note that `\r` is the same under both ASCII and EBCDIC):

```
print "Content-type: text/html\r\n\r\n";
```

The value of `$^O` on OS/390 is "os390".

Some simple tricks for determining if you are running on an EBCDIC platform could include any of the following (perhaps all):

```
if ("\t" eq "\05")   { print "EBCDIC may be spoken here!\n"; }

if (ord('A') == 193) { print "EBCDIC may be spoken here!\n"; }

if (chr(169) eq 'z') { print "EBCDIC may be spoken here!\n"; }
```

Note that one thing you may not want to rely on is the EBCDIC encoding of punctuation characters since these may differ from code page to code page (and once your module or script is rumoured to work with EBCDIC, folks will want it to work with all EBCDIC character sets).

Also see:

perl-mvs list
The perl-mvs@perl.org list is for discussion of porting issues as well as general usage issues for all EBCDIC Perls. Send a message body of "subscribe perl-mvs" to majordomo@perl.org.

AS/400 Perl information at `http://as400.rochester.ibm.com`

### Other perls

Perl has been ported to a variety of platforms that do not fit into any of the above categories. Some, such as AmigaOS, BeOS, MPE/iX, OS/390 (MVS), QNX, and Plan 9, have been well integrated into the standard Perl source code kit. You may need to see the *ports/* directory on CPAN for information, and possibly binaries, for the likes of: acorn, aos, atari, lynxos, riscos, Tandem Guardian, vos, *etc.* (yes we know that some of these OSes may fall under the Unix category but we are not a standards body.)

See also:

Atari, Guido Flohr's page `http://stud.uni-sb.de/~gufl0000/`

HP 300 MPE/iX `http://www.cccd.edu/~markb/perlix.html`

Novell Netware
: A free Perl 5 based PERL.NLM for Novell Netware is available from `http://www.novell.com/`

## FUNCTION IMPLEMENTATIONS

Listed below are functions unimplemented or implemented differently on various platforms. Following each description will be, in parentheses, a list of platforms that the description applies to.

The list may very well be incomplete, or wrong in some places. When in doubt, consult the platform-specific README files in the Perl source distribution, and other documentation resources for a given port.

Be aware, moreover, that even among Unix-ish systems there are variations, and not all functions listed here are necessarily available, though most usually are.

For many functions, you can also query `%Config`, exported by default from `Config.pm`. For example, to check if the platform has the `lstat` call, check `$Config{'d_lstat'}`. See the *Config* manpage for a full description of available variables.

### Alphabetical Listing of Perl Functions

-X FILEHANDLE

-X EXPR

-X
: `-r`, `-w`, and `-x` have only a very limited meaning; directories and applications are executable, and there are no uid/gid considerations. `-o` is not supported. (Mac OS)

`-r`, `-w`, `-x`, and `-o` tell whether or not file is accessible, which may not reflect UIC–based file protections. (VMS)

`-R`, `-W`, `-X`, `-O` are indistinguishable from `-r`, `-w`, `-x`, `-o`. (Mac OS, Win32, VMS)

`-b`, `-c`, `-k`, `-g`, `-p`, `-u`, `-A` are not implemented. (Mac OS)

`-g`, `-k`, `-l`, `-p`, `-u`, `-A` are not particularly meaningful. (Win32, VMS)

`-d` is true if passed a device spec without an explicit directory. (VMS)

`-T` and `-B` are implemented, but might misclassify Mac text files with foreign characters; this is the case will all platforms, but affects Mac OS a lot. (Mac OS)

`-x` (or `-X`) determine if a file ends in one of the executable suffixes. `-S` is meaningless. (Win32)

binmode FILEHANDLE
: Meaningless. (Mac OS)

Reopens file and restores pointer; if function fails, underlying filehandle may be closed, or pointer may be in a different position. (VMS)

The value returned by `tell` may be affected after the call, and the filehandle may be flushed. (Win32)

chmod LIST
: Only limited meaning. Disabling/enabling write permission is mapped to locking/unlocking the file. (Mac OS)

  Only good for changing ‘‘owner’’ read-write access, ‘‘group’’, and ‘‘other’’ bits are meaningless. (Win32)

chown LIST
: Not implemented. (Mac OS, Win32, Plan9)

  Does nothing, but won’t fail. (Win32)

chroot FILENAME

chroot Not implemented. (Mac OS, Win32, VMS, Plan9)

crypt PLAINTEXT,SALT
: May not be available if library or source was not provided when building perl. (Win32)

dbmclose HASH
: Not implemented. (VMS, Plan9)

dbmopen HASH,DBNAME,MODE
: Not implemented. (VMS, Plan9)

dump LABEL
: Not useful. (Mac OS)

  Not implemented. (Win32)

  Invokes VMS debugger. (VMS)

exec LIST
: Not implemented. (Mac OS)

fcntl FILEHANDLE,FUNCTION,SCALAR
: Not implemented. (Win32, VMS)

flock FILEHANDLE,OPERATION
: Not implemented (Mac OS, VMS).

  Available only on Windows NT (not on Windows 95). (Win32)

fork Not implemented. (Mac OS, Win32, AmigaOS)

getlogin Not implemented. (Mac OS)

getpgrp PID
: Not implemented. (Mac OS, Win32, VMS)

getppid Not implemented. (Mac OS, Win32, VMS)

getpriority WHICH,WHO
: Not implemented. (Mac OS, Win32, VMS)

getpwnam NAME
: Not implemented. (Mac OS, Win32)

getgrnam NAME
: Not implemented. (Mac OS, Win32, VMS)

getnetbyname NAME
: Not implemented. (Mac OS, Win32, Plan9)

getpwuid UID
: Not implemented. (Mac OS, Win32)

getgrgid GID
: Not implemented. (Mac OS, Win32, VMS)

getnetbyaddr ADDR,ADDRTYPE
: Not implemented. (Mac OS, Win32, Plan9)

getprotobynumber NUMBER
: Not implemented. (Mac OS)

getservbyport PORT,PROTO
: Not implemented. (Mac OS)

getpwent
: Not implemented. (Mac OS, Win32)

getgrent
: Not implemented. (Mac OS, Win32, VMS)

gethostent
: Not implemented. (Mac OS, Win32)

getnetent
: Not implemented. (Mac OS, Win32, Plan9)

getprotoent
: Not implemented. (Mac OS, Win32, Plan9)

getservent
: Not implemented. (Win32, Plan9)

setpwent
: Not implemented. (Mac OS, Win32)

setgrent
: Not implemented. (Mac OS, Win32, VMS)

sethostent STAYOPEN
: Not implemented. (Mac OS, Win32, Plan9)

setnetent STAYOPEN
: Not implemented. (Mac OS, Win32, Plan9)

setprotoent STAYOPEN
: Not implemented. (Mac OS, Win32, Plan9)

setservent STAYOPEN
: Not implemented. (Plan9, Win32)

endpwent
: Not implemented. (Mac OS, Win32)

endgrent
: Not implemented. (Mac OS, Win32, VMS)

endhostent
: Not implemented. (Mac OS, Win32)

endnetent
: Not implemented. (Mac OS, Win32, Plan9)

endprotoent
: Not implemented. (Mac OS, Win32, Plan9)

endservent
: Not implemented. (Plan9, Win32)

getsockopt SOCKET,LEVEL,OPTNAME
: Not implemented. (Mac OS, Plan9)

glob EXPR

glob Globbing built-in, but only `*` and `?` metacharacters are supported. (Mac OS)

Features depend on external perlglob.exe or perlglob.bat. May be overridden with something like File::DosGlob, which is recommended. (Win32)

ioctl FILEHANDLE,FUNCTION,SCALAR
Not implemented. (VMS)

Available only for socket handles, and it does what the *ioctlsocket()* call in the Winsock API does. (Win32)

kill LIST Not implemented. (Mac OS)

Available only for process handles returned by the `system(1, ...)` method of spawning a process. (Win32)

link OLDFILE,NEWFILE
Not implemented. (Mac OS, Win32, VMS)

lstat FILEHANDLE

lstat EXPR

lstat Not implemented. (VMS)

Return values may be bogus. (Win32)

msgctl ID,CMD,ARG

msgget KEY,FLAGS

msgsnd ID,MSG,FLAGS

msgrcv ID,VAR,SIZE,TYPE,FLAGS
Not implemented. (Mac OS, Win32, VMS, Plan9)

open FILEHANDLE,EXPR

open FILEHANDLE
The `|` variants are only supported if ToolServer is installed. (Mac OS)

open to `|-` and `-|` are unsupported. (Mac OS, Win32)

pipe READHANDLE,WRITEHANDLE
Not implemented. (Mac OS)

readlink EXPR

readlink Not implemented. (Win32, VMS)

select RBITS,WBITS,EBITS,TIMEOUT
Only implemented on sockets. (Win32)

semctl ID,SEMNUM,CMD,ARG

semget KEY,NSEMS,FLAGS

semop KEY,OPSTRING
Not implemented. (Mac OS, Win32, VMS)

setpgrp PID,PGRP
Not implemented. (Mac OS, Win32, VMS)

setpriority WHICH,WHO,PRIORITY
Not implemented. (Mac OS, Win32, VMS)

setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL
: Not implemented. (Mac OS, Plan9)

shmctl ID,CMD,ARG

shmget KEY,SIZE,FLAGS

shmread ID,VAR,POS,SIZE

shmwrite ID,STRING,POS,SIZE
: Not implemented. (Mac OS, Win32, VMS)

socketpair SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL
: Not implemented. (Mac OS, Win32, VMS)

stat FILEHANDLE

stat EXPR

stat
: mtime and atime are the same thing, and ctime is creation time instead of inode change time. (Mac OS)

  device and inode are not meaningful. (Win32)

  device and inode are not necessarily reliable. (VMS)

symlink OLDFILE,NEWFILE
: Not implemented. (Win32, VMS)

syscall LIST
: Not implemented. (Mac OS, Win32, VMS)

system LIST
: Only implemented if ToolServer is installed. (Mac OS)

  As an optimization, may not call the command shell specified in `$ENV{PERL5SHELL}`. `system(1, @args)` spawns an external process and immediately returns its process designator, without waiting for it to terminate. Return value may be used subsequently in `wait` or `waitpid`. (Win32)

times
: Only the first entry returned is nonzero. (Mac OS)

  ‘‘cumulative’’ times will be bogus. On anything other than Windows NT, ‘‘system’’ time will be bogus, and ‘‘user’’ time is actually the time returned by the *clock()* function in the C runtime library. (Win32)

truncate FILEHANDLE,LENGTH

truncate EXPR,LENGTH
: Not implemented. (VMS)

umask EXPR

umask
: Returns undef where unavailable, as of version 5.005.

utime LIST
: Only the modification time is updated. (Mac OS, VMS)

  May not behave as expected. (Win32)

wait

waitpid PID,FLAGS
: Not implemented. (Mac OS)

  Can only be applied to process handles returned for processes spawned using `system(1, ...)`. (Win32)

**AUTHORS / CONTRIBUTORS**

Chris Nandor <pudge@pobox.com>, Gurusamy Sarathy <gsar@umich.edu>, Peter Prymmer <pvhp@forte.com>, Tom Christiansen <tchrist@perl.com>, Nathan Torkington <gnat@frii.com>, Paul Moore <Paul.Moore@uk.origin-it.com>, Matthias Neercher <neeri@iis.ee.ethz.ch>, Charles Bailey <bailey@genetics.upenn.edu>, Luther Huffman <lutherh@stratcom.com>, Gary Ng <71564.1743@CompuServe.COM>, Nick Ing-Simmons <nick@ni-s.u-net.com>, Paul J. Schinder <schinder@pobox.com>, Tom Phoenix <rootbeer@teleport.com>, Hugo van der Sanden <h.sanden@elsevier.nl>, Dominic Dunlop <domo@vo.lu>, Dan Sugalski <sugalskd@ous.edu>, Andreas J. Koenig <koenig@kulturbox.de>, Andrew M. Langmead <aml@world.std.com>, Andy Dougherty <doughera@lafcol.lafayette.edu>, Abigail <abigail@fnx.com>.

This document is maintained by Chris Nandor.

**VERSION**

Version 1.23, last modified 10 July 1998.

**NAME**

perlstyle – Perl style guide

**DESCRIPTION**

Each programmer will, of course, have his or her own preferences in regards to formatting, but there are some general guidelines that will make your programs easier to read, understand, and maintain.

The most important thing is to run your programs under the **–w** flag at all times. You may turn it off explicitly for particular portions of code via the `$^W` variable if you must. You should also always run under `use strict` or know the reason why not. The `use sigtrap` and even `use diagnostics` pragmas may also prove useful.

Regarding aesthetics of code lay out, about the only thing Larry cares strongly about is that the closing curly brace of a multi-line BLOCK should line up with the keyword that started the construct. Beyond that, he has other preferences that aren't so strong:

- 4-column indent.
- Opening curly on same line as keyword, if possible, otherwise line up.
- Space before the opening curly of a multi-line BLOCK.
- One-line BLOCK may be put on one line, including curlies.
- No space before the semicolon.
- Semicolon omitted in "short" one-line BLOCK.
- Space around most operators.
- Space around a "complex" subscript (inside brackets).
- Blank lines between chunks that do different things.
- Uncuddled elses.
- No space between function name and its opening parenthesis.
- Space after each comma.
- Long lines broken after an operator (except "and" and "or").
- Space after last parenthesis matching on current line.
- Line up corresponding items vertically.
- Omit redundant punctuation as long as clarity doesn't suffer.

Larry has his reasons for each of these things, but he doesn't claim that everyone else's mind works the same as his does.

Here are some other more substantive style issues to think about:

- Just because you *CAN* do something a particular way doesn't mean that you *SHOULD* do it that way. Perl is designed to give you several ways to do anything, so consider picking the most readable one. For instance

  ```
  open(FOO,$foo) || die "Can't open $foo: $!";
  ```

  is better than

  ```
  die "Can't open $foo: $!" unless open(FOO,$foo);
  ```

  because the second way hides the main point of the statement in a modifier. On the other hand

  ```
  print "Starting analysis\n" if $verbose;
  ```

  is better than

```
$verbose && print "Starting analysis\n";
```

because the main point isn't whether the user typed **–v** or not.

Similarly, just because an operator lets you assume default arguments doesn't mean that you have to make use of the defaults. The defaults are there for lazy systems programmers writing one-shot programs. If you want your program to be readable, consider supplying the argument.

Along the same lines, just because you *CAN* omit parentheses in many places doesn't mean that you ought to:

```
return print reverse sort num values %array;
return print(reverse(sort num (values(%array))));
```

When in doubt, parenthesize. At the very least it will let some poor schmuck bounce on the % key in **vi**.

Even if you aren't in doubt, consider the mental welfare of the person who has to maintain the code after you, and who will probably put parentheses in the wrong place.

- Don't go through silly contortions to exit a loop at the top or the bottom, when Perl provides the `last` operator so you can exit in the middle. Just "outdent" it a little to make it more visible:

```
LINE:
    for (;;) {
        statements;
      last LINE if $foo;
        next LINE if /^#/;
        statements;
    }
```

- Don't be afraid to use loop labels— they're there to enhance readability as well as to allow multilevel loop breaks. See the previous example.
- Avoid using *grep()* (or *map()*) or 'backticks' in a void context, that is, when you just throw away their return values. Those functions all have return values, so use them. Otherwise use a *foreach()* loop or the *system()* function instead.
- For portability, when using features that may not be implemented on every machine, test the construct in an eval to see if it fails. If you know what version or patchlevel a particular feature was implemented, you can test `$]` (`$PERL_VERSION` in `English`) to see if it will be there. The `Config` module will also let you interrogate values determined by the **Configure** program when Perl was installed.
- Choose mnemonic identifiers. If you can't remember what mnemonic means, you've got a problem.
- While short identifiers like `$gotit` are probably ok, use underscores to separate words. It is generally easier to read `$var_names_like_this` than `$VarNamesLikeThis`, especially for non-native speakers of English. It's also a simple rule that works consistently with VAR_NAMES_LIKE_THIS.

  Package names are sometimes an exception to this rule. Perl informally reserves lowercase module names for "pragma" modules like `integer` and `strict`. Other modules should begin with a capital letter and use mixed case, but probably without underscores due to limitations in primitive file systems' representations of module names as files that must fit into a few sparse bytes.
- You may find it helpful to use letter case to indicate the scope or nature of a variable. For example:

```
$ALL_CAPS_HERE   constants only (beware clashes with perl vars!)
$Some_Caps_Here  package-wide global/static
$no_caps_here    function scope my() or local() variables
```

Function and method names seem to work best as all lowercase. E.g., `$obj->`*as_string()*.

You can use a leading underscore to indicate that a variable or function should not be used outside the package that defined it.

- If you have a really hairy regular expression, use the `/x` modifier and put in some whitespace to make it look a little less like line noise. Don't use slash as a delimiter when your regexp has slashes or backslashes.
- Use the new "and" and "or" operators to avoid having to parenthesize list operators so much, and to reduce the incidence of punctuation operators like `&&` and `||`. Call your subroutines as if they were functions or list operators to avoid excessive ampersands and parentheses.
- Use here documents instead of repeated *print()* statements.
- Line up corresponding things vertically, especially if it'd be too long to fit on one line anyway.

```
$IDX = $ST_MTIME;
$IDX = $ST_ATIME       if $opt_u;
$IDX = $ST_CTIME       if $opt_c;
$IDX = $ST_SIZE        if $opt_s;

mkdir $tmpdir, 0700 or die "can't mkdir $tmpdir: $!";
chdir($tmpdir)      or die "can't chdir $tmpdir: $!";
mkdir 'tmp',   0777 or die "can't mkdir $tmpdir/tmp: $!";
```

- Always check the return codes of system calls. Good error messages should go to STDERR, include which program caused the problem, what the failed system call and arguments were, and (VERY IMPORTANT) should contain the standard system error message for what went wrong. Here's a simple but sufficient example:

```
opendir(D, $dir)     or die "can't opendir $dir: $!";
```

- Line up your transliterations when it makes sense:

```
tr [abc]
   [xyz];
```

- Think about reusability. Why waste brainpower on a one-shot when you might want to do something like it again? Consider generalizing your code. Consider writing a module or object class. Consider making your code run cleanly with `use strict` and **−w** in effect. Consider giving away your code. Consider changing your whole world view. Consider... oh, never mind.
- Be consistent.
- Be nice.

**NAME**

perlpod − plain old documentation

**DESCRIPTION**

A pod-to-whatever translator reads a pod file paragraph by paragraph, and translates it to the appropriate output format. There are three kinds of paragraphs: verbatim, command, and ordinary text.

### Verbatim Paragraph

A verbatim paragraph, distinguished by being indented (that is, it starts with space or tab). It should be reproduced exactly, with tabs assumed to be on 8-column boundaries. There are no special formatting escapes, so you can't italicize or anything like that. A \ means \, and nothing else.

### Command Paragraph

All command paragraphs start with ''='', followed by an identifier, followed by arbitrary text that the command can use however it pleases. Currently recognized commands are

```
=head1 heading
=head2 heading
=item text
=over N
=back
=cut
=pod
=for X
=begin X
=end X
```

=pod

=cut

The ''=pod'' directive does nothing beyond telling the compiler to lay off parsing code through the next ''=cut''. It's useful for adding another paragraph to the doc if you're mixing up code and pod a lot.

=head1

=head2

Head1 and head2 produce first and second level headings, with the text in the same paragraph as the ''=headn'' directive forming the heading description.

=over

=back

=item

Item, over, and back require a little more explanation: ''=over'' starts a section specifically for the generation of a list using ''=item'' commands. At the end of your list, use ''=back'' to end it. You will probably want to give ''4'' as the number to ''=over'', as some formatters will use this for indentation. This should probably be a default. Note also that there are some basic rules to using =item: don't use them outside of an =over/=back block, use at least one inside an =over/=back block, you don't _have_ to include the =back if the list just runs off the document, and perhaps most importantly, keep the items consistent: either use ''=item *'' for all of them, to produce bullets, or use ''=item 1.'', ''=item 2.'', etc., to produce numbered lists, or use ''=item foo'', ''=item bar'', etc., i.e., things that looks nothing like bullets or numbers. If you start with bullets or numbers, stick with them, as many formatters use the first ''=item'' type to decide how to format the list.

=for

=begin

=end

For, begin, and end let you include sections that are not interpreted as pod text, but passed directly to particular formatters. A formatter that can utilize that format will use the section, otherwise it will be completely ignored. The directive ``=for'' specifies that the entire next paragraph is in the format indicated by the first word after ``=for'', like this:

```
=for html <br>
 <p> This is a raw HTML paragraph </p>
```

The paired commands ``=begin'' and ``=end'' work very similarly to ``=for'', but instead of only accepting a single paragraph, all text from ``=begin'' to a paragraph with a matching ``=end'' are treated as a particular format.

Here are some examples of how to use these:

```
=begin html

<br>Figure 1.<IMG SRC="figure1.png"><br>

=end html

=begin text

  ---------------
  |  foo        |
  |        bar  |
  ---------------

^^^^ Figure 1. ^^^^

=end text
```

Some format names that formatters currently are known to accept include ``roff'', ``man'', ``latex'', ``tex'', ``text'', and ``html''. (Some formatters will treat some of these as synonyms.)

And don't forget, when using any command, that the command lasts up until the end of the **paragraph**, not the line. Hence in the examples below, you can see the empty lines after each command to end its paragraph.

Some examples of lists include:

```
=over 4

=item *

First item

=item *

Second item

=back

=over 4
```

```
=item Foo()

Description of Foo function

=item Bar()

Description of Bar function

=back
```

### Ordinary Block of Text

It will be filled, and maybe even justified. Certain interior sequences are recognized both here and in commands:

```
I<text>     italicize text, used for emphasis or variables
B<text>     embolden text, used for switches and programs
S<text>     text contains non-breaking spaces
C<code>     literal code
L<name>     A link (cross reference) to name
                L<name>             manual page
                L<name/ident>       item in manual page
                L<name/"sec">       section in other manual page
                L<"sec">            section in this manual page
                                    (the quotes are optional)
                L</"sec">           ditto
            same as above but only 'text' is used for output.
            (Text can not contain the characters '|' or '>')
                L<text|name>
                L<text|name/ident>
                L<text|name/"sec">
                L<text|"sec">
                L<text|/"sec">

F<file>     Used for filenames
X<index>    An index entry
Z<>         A zero-width character
E<escape>   A named character (very similar to HTML escapes)
                E<lt>               A literal <
                E<gt>               A literal >
                (these are optional except in other interior
                 sequences and when preceded by a capital letter)
                E<n>                Character number n (probably in ASCII)
                E<html>             Some non-numeric HTML entity, such
                                    as E<Agrave>
```

### The Intent

That's it. The intent is simplicity, not power. I wanted paragraphs to look like paragraphs (block format), so that they stand out visually, and so that I could run them through fmt easily to reformat them (that's F7 in my version of **vi**). I wanted the translator (and not me) to worry about whether " or ' is a left quote or a right quote within filled text, and I wanted it to leave the quotes alone, dammit, in verbatim mode, so I could slurp in a working program, shift it over 4 spaces, and have it print out, er, verbatim. And presumably in a constant width font.

In particular, you can leave things like this verbatim in your text:

```
Perl
FILEHANDLE
$variable
function()
manpage(3r)
```

Doubtless a few other commands or sequences will need to be added along the way, but I've gotten along surprisingly well with just these.

Note that I'm not at all claiming this to be sufficient for producing a book. I'm just trying to make an idiot-proof common source for nroff, TeX, and other markup languages, as used for online documentation. Translators exist for **pod2man** (that's for *nroff*(1) and *troff*(1)), **pod2text**, **pod2html**, **pod2latex**, and **pod2fm**.

### Embedding Pods in Perl Modules

You can embed pod documentation in your Perl scripts. Start your documentation with a "=head1" command at the beginning, and end it with a "=cut" command. Perl will ignore the pod text. See any of the supplied library modules for examples. If you're going to put your pods at the end of the file, and you're using an __END__ or __DATA__ cut mark, make sure to put an empty line there before the first pod directive.

```
__END__

=head1 NAME

modern - I am a modern module
```

If you had not had that empty line there, then the translators wouldn't have seen it.

### Common Pod Pitfalls

- Pod translators usually will require paragraphs to be separated by completely empty lines. If you have an apparently empty line with some spaces on it, this can cause odd formatting.
- Translators will mostly add wording around a L<> link, so that L<foo(1)> becomes "the *foo*(1) manpage", for example (see **pod2man** for details). Thus, you shouldn't write things like the L<foo> manpage, if you want the translated document to read sensibly.

  If you don need or want total control of the text used for a link in the output use the form L<show this text|foo> instead.
- The script *pod/checkpods.PL* in the Perl source distribution provides skeletal checking for lines that look empty but aren't **only**, but is there as a placeholder until someone writes Pod::Checker. The best way to check your pod is to pass it through one or more translators and proofread the result, or print out the result and proofread that. Some of the problems found may be bugs in the translators, which you may or may not wish to work around.

## SEE ALSO

the *pod2man* manpage and the section on *PODs: Embedded Documentation* in the *perlsyn* manpage

## AUTHOR

Larry Wall

**NAME**

perlbook − Perl book information

**DESCRIPTION**

The Camel Book, officially known as *Programming Perl, Second Edition*, by Larry Wall et al, is the definitive reference work covering nearly all of Perl. You can order it and other Perl books from O'Reilly & Associates, 1-800-998-9938. Local/overseas is +1 707 829 0515. If you can locate an O'Reilly order form, you can also fax to +1 707 829 0104. If you're web-connected, you can even mosey on over to http://www.ora.com/ for an online order form.

Other Perl books from various publishers and authors can be found listed in the *perlfaq3* manpage.

## NAME

perlembed – how to embed perl in your C program

## DESCRIPTION

### PREAMBLE

Do you want to:

**Use C from Perl?**
Read the *perlxstut* manpage, the *perlxs* manpage, the *h2xs* manpage, and the *perlguts* manpage.

**Use a Unix program from Perl?**
Read about back-quotes and about `system` and `exec` in the *perlfunc* manpage.

**Use Perl from Perl?**
Read about the `do` entry in the *perlfunc* manpage and the `eval` entry in the *perlfunc* manpage and the `require` entry in the *perlfunc* manpage and the `use` entry in the *perlfunc* manpage.

**Use C from C?**
Rethink your design.

**Use Perl from C?**
Read on...

### ROADMAP

the section on *Compiling your C program*

the section on *Adding a Perl interpreter to your C program*

the section on *Calling a Perl subroutine from your C program*

the section on *Evaluating a Perl statement from your C program*

the section on *Performing Perl pattern matches and substitutions from your C program*

the section on *Fiddling with the Perl stack from your C program*

the section on *Maintaining a persistent interpreter*

the section on *Maintaining multiple interpreter instances*

the section on *Using Perl modules, which themselves use C libraries, from your C program*

the section on *Embedding Perl under Win32*

### Compiling your C program

If you have trouble compiling the scripts in this documentation, you're not alone. The cardinal rule: COMPILE THE PROGRAMS IN EXACTLY THE SAME WAY THAT YOUR PERL WAS COMPILED. (Sorry for yelling.)

Also, every C program that uses Perl must link in the *perl library*. What's that, you ask? Perl is itself written in C; the perl library is the collection of compiled C programs that were used to create your perl executable (*/usr/bin/perl* or equivalent). (Corollary: you can't use Perl from your C program unless Perl has been compiled on your machine, or installed properly—that's why you shouldn't blithely copy Perl executables from machine to machine without also copying the *lib* directory.)

When you use Perl from C, your C program will—usually—allocate, "run", and deallocate a *PerlInterpreter* object, which is defined by the perl library.

If your copy of Perl is recent enough to contain this documentation (version 5.002 or later), then the perl library (and *EXTERN.h* and *perl.h*, which you'll also need) will reside in a directory that looks like this:

```
/usr/local/lib/perl5/your_architecture_here/CORE
```

or perhaps just

```
/usr/local/lib/perl5/CORE
```

or maybe something like

```
/usr/opt/perl5/CORE
```

Execute this statement for a hint about where to find CORE:

```
perl -MConfig -e 'print $Config{archlib}'
```

Here's how you'd compile the example in the next section, the section on *Adding a Perl interpreter to your C program*, on my Linux box:

```
% gcc -O2 -Dbool=char -DHAS_BOOL -I/usr/local/include
-I/usr/local/lib/perl5/i586-linux/5.003/CORE
-L/usr/local/lib/perl5/i586-linux/5.003/CORE
-o interp interp.c -lperl -lm
```

(That's all one line.) On my DEC Alpha running old 5.003_05, the incantation is a bit different:

```
% cc -O2 -Olimit 2900 -DSTANDARD_C -I/usr/local/include
-I/usr/local/lib/perl5/alpha-dec_osf/5.00305/CORE
-L/usr/local/lib/perl5/alpha-dec_osf/5.00305/CORE -L/usr/local/lib
-D__LANGUAGE_C__ -D_NO_PROTO -o interp interp.c -lperl -lm
```

How can you figure out what to add? Assuming your Perl is post-5.001, execute a `perl -V` command and pay special attention to the "cc" and "ccflags" information.

You'll have to choose the appropriate compiler (*cc*, *gcc*, et al.) for your machine: `perl -MConfig -e 'print $Config{cc}'` will tell you what to use.

You'll also have to choose the appropriate library directory (*/usr/local/lib/...*) for your machine. If your compiler complains that certain functions are undefined, or that it can't locate *−lperl*, then you need to change the path following the `-L`. If it complains that it can't find *EXTERN.h* and *perl.h*, you need to change the path following the `-I`.

You may have to add extra libraries as well. Which ones? Perhaps those printed by

```
perl -MConfig -e 'print $Config{libs}'
```

Provided your perl binary was properly configured and installed the **ExtUtils::Embed** module will determine all of this information for you:

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

If the **ExtUtils::Embed** module isn't part of your Perl distribution, you can retrieve it from http://www.perl.com/perl/CPAN/modules/by-module/ExtUtils::Embed. (If this documentation came from your Perl distribution, then you're running 5.004 or better and you already have it.)

The **ExtUtils::Embed** kit on CPAN also contains all source code for the examples in this document, tests, additional examples and other information you may find useful.

### Adding a Perl interpreter to your C program

In a sense, perl (the C program) is a good example of embedding Perl (the language), so I'll demonstrate embedding with *miniperlmain.c*, included in the source distribution. Here's a bastardized, nonportable version of *miniperlmain.c* containing the essentials of embedding:

```
#include <EXTERN.h>               /* from the Perl distribution     */
#include <perl.h>                 /* from the Perl distribution     */
```

```
static PerlInterpreter *my_perl;  /***    The Perl interpreter    ***/

int main(int argc, char **argv, char **env)
{
    my_perl = perl_alloc();
    perl_construct(my_perl);
    perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
    perl_run(my_perl);
    perl_destruct(my_perl);
    perl_free(my_perl);
}
```

Notice that we don't use the `env` pointer. Normally handed to `perl_parse` as its final argument, `env` here is replaced by `NULL`, which means that the current environment will be used.

Now compile this program (I'll call it *interp.c*) into an executable:

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

After a successful compilation, you'll be able to use *interp* just like perl itself:

```
% interp
print "Pretty Good Perl \n";
print "10890 - 9801 is ", 10890 - 9801;
<CTRL-D>
Pretty Good Perl
10890 - 9801 is 1089
```

or

```
% interp -e 'printf("%x", 3735928559)'
deadbeef
```

You can also read and execute Perl statements from a file while in the midst of your C program, by placing the filename in *argv[1]* before calling *perl_run*.

### Calling a Perl subroutine from your C program

To call individual Perl subroutines, you can use any of the **perl_call_*** functions documented in the *perlcall* manpage. In this example we'll use `perl_call_argv`.

That's shown below, in a program I'll call *showtime.c*.

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

int main(int argc, char **argv, char **env)
{
    char *args[] = { NULL };
    my_perl = perl_alloc();
    perl_construct(my_perl);

    perl_parse(my_perl, NULL, argc, argv, NULL);

    /*** skipping perl_run() ***/
```

```
        perl_call_argv("showtime", G_DISCARD | G_NOARGS, args);

        perl_destruct(my_perl);
        perl_free(my_perl);
    }
```

where *showtime* is a Perl subroutine that takes no arguments (that's the *G_NOARGS*) and for which I'll ignore the return value (that's the *G_DISCARD*). Those flags, and others, are discussed in the *perlcall* manpage.

I'll define the *showtime* subroutine in a file called *showtime.pl*:

```
print "I shan't be printed.";

sub showtime {
    print time;
}
```

Simple enough. Now compile and run:

```
% cc -o showtime showtime.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

% showtime showtime.pl
818284590
```

yielding the number of seconds that elapsed between January 1, 1970 (the beginning of the Unix epoch), and the moment I began writing this sentence.

In this particular case we don't have to call *perl_run*, but in general it's considered good practice to ensure proper initialization of library code, including execution of all object `DESTROY` methods and package `END {}` blocks.

If you want to pass arguments to the Perl subroutine, you can add strings to the `NULL`–terminated `args` list passed to *perl_call_argv*. For other data types, or to examine return values, you'll need to manipulate the Perl stack. That's demonstrated in the last section of this document: the section on *Fiddling with the Perl stack from your C program*.

### Evaluating a Perl statement from your C program

Perl provides two API functions to evaluate pieces of Perl code. These are the `perl_eval_sv` entry in the *perlguts* manpage and the `perl_eval_pv` entry in the *perlguts* manpage.

Arguably, these are the only routines you'll ever need to execute snippets of Perl code from within your C program. Your code can be as long as you wish; it can contain multiple statements; it can employ the `use` entry in the *perlfunc* manpage, the `require` entry in the *perlfunc* manpage, and the `do` entry in the *perlfunc* manpage to include external Perl files.

*perl_eval_pv* lets us evaluate individual Perl strings, and then extract variables for coercion into C types. The following program, *string.c*, executes three Perl strings, extracting an `int` from the first, a `float` from the second, and a `char *` from the third.

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

main (int argc, char **argv, char **env)
{
    char *embedding[] = { "", "-e", "0" };

    my_perl = perl_alloc();
    perl_construct( my_perl );

    perl_parse(my_perl, NULL, 3, embedding, NULL);
    perl_run(my_perl);

    /** Treat $a as an integer **/
    perl_eval_pv("$a = 3; $a **= 2", TRUE);
    printf("a = %d\n", SvIV(perl_get_sv("a", FALSE)));

    /** Treat $a as a float **/
    perl_eval_pv("$a = 3.14; $a **= 2", TRUE);
    printf("a = %f\n", SvNV(perl_get_sv("a", FALSE)));

    /** Treat $a as a string **/
    perl_eval_pv("$a = 'rekcaH lreP rehtonA tsuJ'; $a = reverse($a);", TRUE);
    printf("a = %s\n", SvPV(perl_get_sv("a", FALSE), PL_na));

    perl_destruct(my_perl);
    perl_free(my_perl);
}
```

All of those strange functions with *sv* in their names help convert Perl scalars to C types. They're described in the *perlguts* manpage.

If you compile and run *string.c*, you'll see the results of using *SvIV()* to create an `int`, *SvNV()* to create a `float`, and *SvPV()* to create a string:

```
a = 9
a = 9.859600
a = Just Another Perl Hacker
```

In the example above, we've created a global variable to temporarily store the computed value of our eval'd expression. It is also possible and in most cases a better strategy to fetch the return value from *perl_eval_pv()* instead. Example:

```
...
SV *val = perl_eval_pv("reverse 'rekcaH lreP rehtonA tsuJ'", TRUE);
printf("%s\n", SvPV(val,PL_na));
...
```

This way, we avoid namespace pollution by not creating global variables and we've simplified our code as well.

**Performing Perl pattern matches and substitutions from your C program**

The *perl_eval_sv()* function lets us evaluate strings of Perl code, so we can define some functions that use it to "specialize" in matches and substitutions: *match()*, *substitute()*, and *matches()*.

```
I32 match(SV *string, char *pattern);
```

Given a string and a pattern (e.g., `m/clasp/` or `/\b\w*\b/`, which in your C program might appear as "/\\b\\w*\\b/"), *match()* returns 1 if the string matches the pattern and 0 otherwise.

```
int substitute(SV **string, char *pattern);
```

Given a pointer to an `SV` and an `=~` operation (e.g., `s/bob/robert/g` or `tr[A-Z][a-z]`), *substitute()* modifies the string within the `AV` at according to the operation, returning the number of substitutions made.

```
int matches(SV *string, char *pattern, AV **matches);
```

Given an `SV`, a pattern, and a pointer to an empty `AV`, *matches()* evaluates `$string =~ $pattern` in an array context, and fills in *matches* with the array elements, returning the number of matches found.

Here's a sample program, *match.c*, that uses all three (long lines have been wrapped here):

```
#include <EXTERN.h>
#include <perl.h>

/** my_perl_eval_sv(code, error_check)
** kinda like perl_eval_sv(),
** but we pop the return value off the stack
**/
SV* my_perl_eval_sv(SV *sv, I32 croak_on_error)
{
    dSP;
    SV* retval;

    PUSHMARK(SP);
    perl_eval_sv(sv, G_SCALAR);

    SPAGAIN;
    retval = POPs;
    PUTBACK;

    if (croak_on_error && SvTRUE(ERRSV))
       croak(SvPVx(ERRSV, PL_na));

    return retval;
}

/** match(string, pattern)
**
** Used for matches in a scalar context.
**
** Returns 1 if the match was successful; 0 otherwise.
**/

I32 match(SV *string, char *pattern)
{
    SV *command = NEWSV(1099, 0), *retval;

    sv_setpvf(command, "my $string = '%s'; $string =~ %s",
             SvPV(string,PL_na), pattern);

    retval = my_perl_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    return SvIV(retval);
}

/** substitute(string, pattern)
**
** Used for =~ operations that modify their left-hand side (s/// and tr///)
**
** Returns the number of successful matches, and
** modifies the input string if there were any.
**/

I32 substitute(SV **string, char *pattern)
```

```
{
    SV *command = NEWSV(1099, 0), *retval;

    sv_setpvf(command, "$string = '%s'; ($string =~ %s)",
             SvPV(*string,PL_na), pattern);

    retval = my_perl_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    *string = perl_get_sv("string", FALSE);
    return SvIV(retval);
}

/** matches(string, pattern, matches)
**
** Used for matches in an array context.
**
** Returns the number of matches,
** and fills in **matches with the matching substrings
**/

I32 matches(SV *string, char *pattern, AV **match_list)
{
    SV *command = NEWSV(1099, 0);
    I32 num_matches;

    sv_setpvf(command, "my $string = '%s'; @array = ($string =~ %s)",
             SvPV(string,PL_na), pattern);

    my_perl_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    *match_list = perl_get_av("array", FALSE);
    num_matches = av_len(*match_list) + 1; /** assume $[ is 0 **/

    return num_matches;
}

main (int argc, char **argv, char **env)
{
    PerlInterpreter *my_perl = perl_alloc();
    char *embedding[] = { "", "-e", "0" };
    AV *match_list;
    I32 num_matches, i;
    SV *text = NEWSV(1099,0);

    perl_construct(my_perl);
    perl_parse(my_perl, NULL, 3, embedding, NULL);

    sv_setpv(text, "When he is at a convenience store and the bill comes to some am

    if (match(text, "m/quarter/")) /** Does text contain 'quarter'? **/
       printf("match: Text contains the word 'quarter'.\n\n");
    else
```

```
        printf("match: Text doesn't contain the word 'quarter'.\n\n");

     if (match(text, "m/eighth/")) /** Does text contain 'eighth'? **/
        printf("match: Text contains the word 'eighth'.\n\n");
     else
        printf("match: Text doesn't contain the word 'eighth'.\n\n");

     /** Match all occurrences of /wi../ **/
     num_matches = matches(text, "m/(wi..)/g", &match_list);
     printf("matches: m/(wi..)/g found %d matches...\n", num_matches);

     for (i = 0; i < num_matches; i++)
        printf("match: %s\n", SvPV(*av_fetch(match_list, i, FALSE),PL_na));
     printf("\n");

     /** Remove all vowels from text **/
     num_matches = substitute(&text, "s/[aeiou]//gi");
     if (num_matches) {
        printf("substitute: s/[aeiou]//gi...%d substitutions made.\n",
               num_matches);
        printf("Now text is: %s\n\n", SvPV(text,PL_na));
     }

     /** Attempt a substitution **/
     if (!substitute(&text, "s/Perl/C/")) {
        printf("substitute: s/Perl/C...No substitution made.\n\n");
     }

     SvREFCNT_dec(text);
     PL_perl_destruct_level = 1;
     perl_destruct(my_perl);
     perl_free(my_perl);
 }
```

which produces the output (again, long lines have been wrapped here)

```
match: Text contains the word 'quarter'.

match: Text doesn't contain the word 'eighth'.

matches: m/(wi..)/g found 2 matches...
match: will
match: with

substitute: s/[aeiou]//gi...139 substitutions made.
Now text is: Whn h s t  cnvnnc str nd th bll cms t sm mnt lk 76 cnts,
Mynrd s wr tht thr s smthng h *shld* d, smthng tht wll nbl hm t gt bck
qrtr, bt h hs n d *wht*.  H fmbls thrgh hs rd sqzy chngprs nd gvs th by
thr xtr pnns wth hs dllr, hpng tht h mght lck nt th crrct mnt.  Th by gvs
hm bck tw f hs wn pnns nd thn th bg shny qrtr tht s hs prz. -RCHH

substitute: s/Perl/C...No substitution made.
```

**Fiddling with the Perl stack from your C program**

When trying to explain stacks, most computer science textbooks mumble something about spring-loaded columns of cafeteria plates: the last thing you pushed on the stack is the first thing you pop off. That'll do for our purposes: your C program will push some arguments onto "the Perl stack", shut its eyes while some magic happens, and then pop the results— the return value of your Perl subroutine— off the stack.

First you'll need to know how to convert between C types and Perl types, with *newSViv()* and *sv_setnv()* and *newAV()* and all their friends. They're described in the *perlguts* manpage.

Then you'll need to know how to manipulate the Perl stack. That's described in the *perlcall* manpage.

Once you've understood those, embedding Perl in C is easy.

Because C has no builtin function for integer exponentiation, let's make Perl's ** operator available to it (this is less useful than it sounds, because Perl implements ** with C's *pow()* function). First I'll create a stub exponentiation function in *power.pl*:

```
sub expo {
    my ($a, $b) = @_;
    return $a ** $b;
}
```

Now I'll create a C program, *power.c*, with a function *PerlPower()* that contains all the perlguts necessary to push the two arguments into *expo()* and to pop the return value out. Take a deep breath...

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

static void
PerlPower(int a, int b)
{
  dSP;                            /* initialize stack pointer      */
  ENTER;                          /* everything created after here */
  SAVETMPS;                       /* ...is a temporary variable.   */
  PUSHMARK(SP);                   /* remember the stack pointer    */
  XPUSHs(sv_2mortal(newSViv(a))); /* push the base onto the stack  */
  XPUSHs(sv_2mortal(newSViv(b))); /* push the exponent onto stack  */
  PUTBACK;                      /* make local stack pointer global */
  perl_call_pv("expo", G_SCALAR); /* call the function             */
  SPAGAIN;                        /* refresh stack pointer         */
                                /* pop the return value from stack */
  printf ("%d to the %dth power is %d.\n", a, b, POPi);
  PUTBACK;
  FREETMPS;                       /* free that return value        */
  LEAVE;                       /* ...and the XPUSHed "mortal" args.*/
}

int main (int argc, char **argv, char **env)
{
  char *my_argv[] = { "", "power.pl" };

  my_perl = perl_alloc();
  perl_construct( my_perl );
```

```
      perl_parse(my_perl, NULL, 2, my_argv, (char **)NULL);
      perl_run(my_perl);

      PerlPower(3, 4);                      /*** Compute 3 ** 4 ***/

      perl_destruct(my_perl);
      perl_free(my_perl);
    }
```

Compile and run:

```
    % cc -o power power.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

    % power
    3 to the 4th power is 81.
```

**Maintaining a persistent interpreter**

When developing interactive and/or potentially long-running applications, it's a good idea to maintain a persistent interpreter rather than allocating and constructing a new interpreter multiple times. The major reason is speed: since Perl will only be loaded into memory once.

However, you have to be more cautious with namespace and variable scoping when using a persistent interpreter. In previous examples we've been using global variables in the default package `main`. We knew exactly what code would be run, and assumed we could avoid variable collisions and outrageous symbol table growth.

Let's say your application is a server that will occasionally run Perl code from some arbitrary file. Your server has no way of knowing what code it's going to run. Very dangerous.

If the file is pulled in by `perl_parse()`, compiled into a newly constructed interpreter, and subsequently cleaned out with `perl_destruct()` afterwards, you're shielded from most namespace troubles.

One way to avoid namespace collisions in this scenario is to translate the filename into a guaranteed-unique package name, and then compile the code into that package using the `eval` entry in the *perlfunc* manpage. In the example below, each file will only be compiled once. Or, the application might choose to clean out the symbol table associated with the file after it's no longer needed. Using the `perl_call_argv` entry in the *perlcall* manpage, We'll call the subroutine `Embed::Persistent::eval_file` which lives in the file `persistent.pl` and pass the filename and boolean cleanup/cache flag as arguments.

Note that the process will continue to grow for each file that it uses. In addition, there might be `AUTOLOAD`ed subroutines and other conditions that cause Perl's symbol table to grow. You might want to add some logic that keeps track of the process size, or restarts itself after a certain number of requests, to ensure that memory consumption is minimized. You'll also want to scope your variables with the `my` entry in the *perlfunc* manpage whenever possible.

```
 package Embed::Persistent;
 #persistent.pl

 use strict;
 use vars '%Cache';
 use Symbol qw(delete_package);
```

```
sub valid_package_name {
    my($string) = @_;
    $string =~ s/([^A-Za-z0-9\/])/sprintf("_%2x",unpack("C",$1))/eg;
    # second pass only for words starting with a digit
    $string =~ s|/(\d)|sprintf("/_%2x",unpack("C",$1))|eg;

    # Dress it up as a real package name
    $string =~ s|/|::|g;
    return "Embed" . $string;
}

sub eval_file {
    my($filename, $delete) = @_;
    my $package = valid_package_name($filename);
    my $mtime = -M $filename;
    if(defined $Cache{$package}{mtime}
       &&
       $Cache{$package}{mtime} <= $mtime)
    {
       # we have compiled this subroutine already,
       # it has not been updated on disk, nothing left to do
       print STDERR "already compiled $package->handler\n";
    }
    else {
       local *FH;
       open FH, $filename or die "open '$filename' $!";
       local($/) = undef;
       my $sub = <FH>;
       close FH;

       #wrap the code into a subroutine inside our unique package
       my $eval = qq{package $package; sub handler { $sub; }};
       {
           # hide our variables within this block
           my($filename,$mtime,$package,$sub);
           eval $eval;
       }
       die $@ if $@;

       #cache it unless we're cleaning out each time
       $Cache{$package}{mtime} = $mtime unless $delete;
    }

    eval {$package->handler;};
    die $@ if $@;

    delete_package($package) if $delete;

    #take a look if you want
    #print Devel::Symdump->rnew($package)->as_string, $/;
}

1;
```

```
__END__

/* persistent.c */
#include <EXTERN.h>
#include <perl.h>

/* 1 = clean out filename's symbol table after each request, 0 = don't */
#ifndef DO_CLEAN
#define DO_CLEAN 0
#endif

static PerlInterpreter *perl = NULL;

int
main(int argc, char **argv, char **env)
{
    char *embedding[] = { "", "persistent.pl" };
    char *args[] = { "", DO_CLEAN, NULL };
    char filename [1024];
    int exitstatus = 0;

    if((perl = perl_alloc()) == NULL) {
       fprintf(stderr, "no memory!");
       exit(1);
    }
    perl_construct(perl);

    exitstatus = perl_parse(perl, NULL, 2, embedding, NULL);

    if(!exitstatus) {
       exitstatus = perl_run(perl);

       while(printf("Enter file name: ") && gets(filename)) {

           /* call the subroutine, passing it the filename as an argument */
           args[0] = filename;
           perl_call_argv("Embed::Persistent::eval_file",
                          G_DISCARD | G_EVAL, args);

           /* check $@ */
           if(SvTRUE(ERRSV))
               fprintf(stderr, "eval error: %s\n", SvPV(ERRSV,PL_na));
       }
    }

    PL_perl_destruct_level = 0;
    perl_destruct(perl);
    perl_free(perl);
    exit(exitstatus);
}
```

Now compile:

```
% cc -o persistent persistent.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

Here's a example script file:

```
#test.pl
my $string = "hello";
foo($string);

sub foo {
    print "foo says: @_\n";
}
```

Now run:

```
% persistent
Enter file name: test.pl
foo says: hello
Enter file name: test.pl
already compiled Embed::test_2epl->handler
foo says: hello
Enter file name: ^C
```

**Maintaining multiple interpreter instances**

Some rare applications will need to create more than one interpreter during a session. Such an application might sporadically decide to release any resources associated with the interpreter.

The program must take care to ensure that this takes place *before* the next interpreter is constructed. By default, the global variable `PL_perl_destruct_level` is set to `0`, since extra cleaning isn't needed when a program has only one interpreter.

Setting `PL_perl_destruct_level` to `1` makes everything squeaky clean:

```
PL_perl_destruct_level = 1;

while(1) {
    ...
    /* reset global variables here with PL_perl_destruct_level = 1 */
    perl_construct(my_perl);
    ...
    /* clean and reset _everything_ during perl_destruct */
    perl_destruct(my_perl);
    perl_free(my_perl);
    ...
    /* let's go do it again! */
}
```

When *perl_destruct()* is called, the interpreter's syntax parse tree and symbol tables are cleaned up, and global variables are reset.

Now suppose we have more than one interpreter instance running at the same time. This is feasible, but only if you used the `-DMULTIPLICITY` flag when building Perl. By default, that sets `PL_perl_destruct_level` to `1`.

Let's give it a try:

```
#include <EXTERN.h>
#include <perl.h>
```

```
/* we're going to embed two interpreters */
/* we're going to embed two interpreters */

#define SAY_HELLO "-e", "print qq(Hi, I'm $^X\n)"

int main(int argc, char **argv, char **env)
{
    PerlInterpreter
        *one_perl = perl_alloc(),
        *two_perl = perl_alloc();
    char *one_args[] = { "one_perl", SAY_HELLO };
    char *two_args[] = { "two_perl", SAY_HELLO };

    perl_construct(one_perl);
    perl_construct(two_perl);

    perl_parse(one_perl, NULL, 3, one_args, (char **)NULL);
    perl_parse(two_perl, NULL, 3, two_args, (char **)NULL);

    perl_run(one_perl);
    perl_run(two_perl);

    perl_destruct(one_perl);
    perl_destruct(two_perl);

    perl_free(one_perl);
    perl_free(two_perl);
}
```

Compile as usual:

```
% cc -o multiplicity multiplicity.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

Run it, Run it:

```
% multiplicity
Hi, I'm one_perl
Hi, I'm two_perl
```

**Using Perl modules, which themselves use C libraries, from your C program**

If you've played with the examples above and tried to embed a script that *use()*s a Perl module (such as *Socket*) which itself uses a C or C++ library, this probably happened:

```
Can't load module Socket, dynamic loading not available in this perl.
  (You may need to build a new perl executable which either supports
  dynamic loading or has the Socket module statically linked into it.)
```

What's wrong?

Your interpreter doesn't know how to communicate with these extensions on its own. A little glue will help. Up until now you've been calling *perl_parse()*, handing it NULL for the second argument:

```
perl_parse(my_perl, NULL, argc, my_argv, NULL);
```

That's where the glue code can be inserted to create the initial contact between Perl and linked C/C++ routines. Let's take a look some pieces of *perlmain.c* to see how Perl does this:

```
#ifdef __cplusplus
#  define EXTERN_C extern "C"
#else
#  define EXTERN_C extern
#endif

static void xs_init _((void));

EXTERN_C void boot_DynaLoader _((CV* cv));
EXTERN_C void boot_Socket _((CV* cv));

EXTERN_C void
xs_init()
{
       char *file = __FILE__;
       /* DynaLoader is a special case */
       newXS("DynaLoader::boot_DynaLoader", boot_DynaLoader, file);
       newXS("Socket::bootstrap", boot_Socket, file);
}
```

Simply put: for each extension linked with your Perl executable (determined during its initial configuration on your computer or when adding a new extension), a Perl subroutine is created to incorporate the extension's routines. Normally, that subroutine is named *Module::bootstrap()* and is invoked when you say *use Module*. In turn, this hooks into an XSUB, *boot_Module*, which creates a Perl counterpart for each of the extension's XSUBs. Don't worry about this part; leave that to the *xsubpp* and extension authors. If your extension is dynamically loaded, DynaLoader creates *Module::bootstrap()* for you on the fly. In fact, if you have a working DynaLoader then there is rarely any need to link in any other extensions statically.

Once you have this code, slap it into the second argument of *perl_parse()*:

```
perl_parse(my_perl, xs_init, argc, my_argv, NULL);
```

Then compile:

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

% interp
  use Socket;
  use SomeDynamicallyLoadedModule;

  print "Now I can use extensions!\n"'
```

**ExtUtils::Embed** can also automate writing the *xs_init* glue code.

```
% perl -MExtUtils::Embed -e xsinit -- -o perlxsi.c
% cc -c perlxsi.c `perl -MExtUtils::Embed -e ccopts`
% cc -c interp.c  `perl -MExtUtils::Embed -e ccopts`
% cc -o interp perlxsi.o interp.o `perl -MExtUtils::Embed -e ldopts`
```

Consult the *perlxs* manpage and the *perlguts* manpage for more details.

**Embedding Perl under Win32**

At the time of this writing (5.004), there are two versions of Perl which run under Win32. (The two versions are merging in 5.005.) Interfacing to ActiveState's Perl library is quite different from the examples in this documentation, as significant changes were made to the internal Perl API. However, it is possible to embed ActiveState's Perl runtime. For details, see the Perl for Win32 FAQ at http://www.perl.com/perl/faq/win32/Perl_for_Win32_FAQ.html.

With the ''official'' Perl version 5.004 or higher, all the examples within this documentation will compile and run untouched, although the build process is slightly different between Unix and Win32.

For starters, backticks don't work under the Win32 native command shell. The ExtUtils::Embed kit on CPAN ships with a script called **genmake**, which generates a simple makefile to build a program from a single C source file. It can be used like this:

```
C:\ExtUtils-Embed\eg> perl genmake interp.c
C:\ExtUtils-Embed\eg> nmake
C:\ExtUtils-Embed\eg> interp -e "print qq{I'm embedded in Win32!\n}"
```

You may wish to use a more robust environment such as the Microsoft Developer Studio. In this case, run this to generate perlxsi.c:

```
perl -MExtUtils::Embed -e xsinit
```

Create a new project and Insert −> Files into Project: perlxsi.c, perl.lib, and your own source files, e.g. interp.c. Typically you'll find perl.lib in **C:\perl\lib\CORE**, if not, you should see the **CORE** directory relative to `perl -V:archlib`. The studio will also need this path so it knows where to find Perl include files. This path can be added via the Tools −> Options −> Directories menu. Finally, select Build −> Build interp.exe and you're ready to go.

### MORAL

You can sometimes *write faster code* in C, but you can always *write code faster* in Perl. Because you can use each from the other, combine them as you wish.

### AUTHOR

Jon Orwant <*orwant@tpj.com*> and Doug MacEachern <*dougm@osf.org*>, with small contributions from Tim Bunce, Tom Christiansen, Guy Decoux, Hallvard Furuseth, Dov Grobgeld, and Ilya Zakharevich.

Doug MacEachern has an article on embedding in Volume 1, Issue 4 of The Perl Journal (http://tpj.com). Doug is also the developer of the most widely-used Perl embedding: the mod_perl system (perl.apache.org), which embeds Perl in the Apache web server. Oracle, Binary Evolution, ActiveState, and Ben Sugars's nsapi_perl have used this model for Oracle, Netscape and Internet Information Server Perl plugins.

July 22, 1998

### COPYRIGHT

**NAME**

perlapio − perl's IO abstraction interface.

**SYNOPSIS**

```
PerlIO *PerlIO_stdin(void);
PerlIO *PerlIO_stdout(void);
PerlIO *PerlIO_stderr(void);

PerlIO *PerlIO_open(const char *,const char *);
int     PerlIO_close(PerlIO *);

int     PerlIO_stdoutf(const char *,...)
int     PerlIO_puts(PerlIO *,const char *);
int     PerlIO_putc(PerlIO *,int);
int     PerlIO_write(PerlIO *,const void *,size_t);
int     PerlIO_printf(PerlIO *, const char *,...);
int     PerlIO_vprintf(PerlIO *, const char *, va_list);
int     PerlIO_flush(PerlIO *);

int     PerlIO_eof(PerlIO *);
int     PerlIO_error(PerlIO *);
void    PerlIO_clearerr(PerlIO *);

int     PerlIO_getc(PerlIO *);
int     PerlIO_ungetc(PerlIO *,int);
int     PerlIO_read(PerlIO *,void *,size_t);

int     PerlIO_fileno(PerlIO *);
PerlIO *PerlIO_fdopen(int, const char *);
PerlIO *PerlIO_importFILE(FILE *, int flags);
FILE   *PerlIO_exportFILE(PerlIO *, int flags);
FILE   *PerlIO_findFILE(PerlIO *);
void    PerlIO_releaseFILE(PerlIO *,FILE *);

void    PerlIO_setlinebuf(PerlIO *);

long    PerlIO_tell(PerlIO *);
int     PerlIO_seek(PerlIO *,off_t,int);
int     PerlIO_getpos(PerlIO *,Fpos_t *)
int     PerlIO_setpos(PerlIO *,Fpos_t *)
void    PerlIO_rewind(PerlIO *);

int     PerlIO_has_base(PerlIO *);
int     PerlIO_has_cntptr(PerlIO *);
int     PerlIO_fast_gets(PerlIO *);
int     PerlIO_canset_cnt(PerlIO *);

char   *PerlIO_get_ptr(PerlIO *);
int     PerlIO_get_cnt(PerlIO *);
void    PerlIO_set_cnt(PerlIO *,int);
void    PerlIO_set_ptrcnt(PerlIO *,char *,int);
char   *PerlIO_get_base(PerlIO *);
int     PerlIO_get_bufsiz(PerlIO *);
```

## DESCRIPTION

Perl's source code should use the above functions instead of those defined in ANSI C's *stdio.h*. The perl headers will `#define` them to the I/O mechanism selected at Configure time.

The functions are modeled on those in *stdio.h*, but parameter order has been "tidied up a little".

**PerlIO ***
: This takes the place of FILE *. Like FILE * it should be treated as opaque (it is probably safe to assume it is a pointer to something).

**PerlIO_stdin()**, **PerlIO_stdout()**, **PerlIO_stderr()**
: Use these rather than `stdin`, `stdout`, `stderr`. They are written to look like "function calls" rather than variables because this makes it easier to *make them* function calls if platform cannot export data to loaded modules, or if (say) different "threads" might have different values.

**PerlIO_open(path, mode)**, **PerlIO_fdopen(fd,mode)**
: These correspond to *fopen()*/*fdopen()* arguments are the same.

**PerlIO_printf(f,fmt,...)**, **PerlIO_vprintf(f,fmt,a)**
: These are *fprintf()*/*vfprintf()* equivalents.

**PerlIO_stdoutf(fmt,...)**
: This is *printf()* equivalent. printf is #defined to this function, so it is (currently) legal to use `printf(fmt,...)` in perl sources.

**PerlIO_read(f,buf,count)**, **PerlIO_write(f,buf,count)**
: These correspond to *fread()* and *fwrite()*. Note that arguments are different, there is only one "count" and order has "file" first.

**PerlIO_close(f)**

**PerlIO_puts(f,s)**, **PerlIO_putc(f,c)**
: These correspond to *fputs()* and *fputc()*. Note that arguments have been revised to have "file" first.

**PerlIO_ungetc(f,c)**
: This corresponds to *ungetc()*. Note that arguments have been revised to have "file" first.

**PerlIO_getc(f)**
: This corresponds to *getc()*.

**PerlIO_eof(f)**
: This corresponds to *feof()*.

**PerlIO_error(f)**
: This corresponds to *ferror()*.

**PerlIO_fileno(f)**
: This corresponds to *fileno()*, note that on some platforms, the meaning of "fileno" may not match Unix.

**PerlIO_clearerr(f)**
: This corresponds to *clearerr()*, i.e., clears 'eof' and 'error' flags for the "stream".

**PerlIO_flush(f)**
: This corresponds to *fflush()*.

**PerlIO_tell(f)**
: This corresponds to *ftell()*.

**PerlIO_seek(f,o,w)**
: This corresponds to *fseek()*.

**PerlIO_getpos(f,p)**, **PerlIO_setpos(f,p)**
: These correspond to *fgetpos()* and *fsetpos()*. If platform does not have the stdio calls then they are implemented in terms of *PerlIO_tell()* and *PerlIO_seek()*.

**PerlIO_rewind(f)**
This corresponds to *rewind()*. Note may be redefined in terms of *PerlIO_seek()* at some point.

**PerlIO_tmpfile()**
This corresponds to *tmpfile()*, i.e., returns an anonymous PerlIO which will automatically be deleted when closed.

### Co-existence with stdio

There is outline support for co-existence of PerlIO with stdio. Obviously if PerlIO is implemented in terms of stdio there is no problem. However if perlio is implemented on top of (say) sfio then mechanisms must exist to create a FILE * which can be passed to library code which is going to use stdio calls.

**PerlIO_importFILE(f,flags)**
Used to get a PerlIO * from a FILE *. May need additional arguments, interface under review.

**PerlIO_exportFILE(f,flags)**
Given an PerlIO * return a 'native' FILE * suitable for passing to code expecting to be compiled and linked with ANSI C *stdio.h*.

The fact that such a FILE * has been 'exported' is recorded, and may affect future PerlIO operations on the original PerlIO *.

**PerlIO_findFILE(f)**
Returns previously 'exported' FILE * (if any). Place holder until interface is fully defined.

**PerlIO_releaseFILE(p,f)**
Calling PerlIO_releaseFILE informs PerlIO that all use of FILE * is complete. It is removed from list of 'exported' FILE *s, and associated PerlIO * should revert to original behaviour.

**PerlIO_setlinebuf(f)**
This corresponds to *setlinebuf()*. Use is deprecated pending further discussion. (Perl core uses it *only* when "dumping"; it has nothing to do with $| auto-flush.)

In addition to user API above there is an "implementation" interface which allows perl to get at internals of PerlIO. The following calls correspond to the various FILE_xxx macros determined by Configure. This section is really of interest to only those concerned with detailed perl-core behaviour or implementing a PerlIO mapping.

**PerlIO_has_cntptr(f)**
Implementation can return pointer to current position in the "buffer" and a count of bytes available in the buffer.

**PerlIO_get_ptr(f)**
Return pointer to next readable byte in buffer.

**PerlIO_get_cnt(f)**
Return count of readable bytes in the buffer.

**PerlIO_canset_cnt(f)**
Implementation can adjust its idea of number of bytes in the buffer.

**PerlIO_fast_gets(f)**
Implementation has all the interfaces required to allow perl's fast code to handle <FILE> mechanism.

```
PerlIO_fast_gets(f) = PerlIO_has_cntptr(f) && \
                      PerlIO_canset_cnt(f) && \
                      'Can set pointer into buffer'
```

**PerlIO_set_ptrcnt(f,p,c)**
Set pointer into buffer, and a count of bytes still in the buffer. Should be used only to set pointer to within range implied by previous calls to `PerlIO_get_ptr` and `PerlIO_get_cnt`.

**PerlIO_set_cnt(f,c)**
Obscure − set count of bytes in the buffer. Deprecated. Currently used in only doio.c to force count < −1 to −1. Perhaps should be PerlIO_set_empty or similar. This call may actually do nothing if "count" is deduced from pointer and a "limit".

**PerlIO_has_base(f)**
Implementation has a buffer, and can return pointer to whole buffer and its size. Used by perl for **−T** / **−B** tests. Other uses would be very obscure...

**PerlIO_get_base(f)**
Return *start* of buffer.

**PerlIO_get_bufsiz(f)**
Return *total size* of buffer.

## NAME

perlxs – XS language reference manual

## DESCRIPTION

### Introduction

XS is a language used to create an extension interface between Perl and some C library which one wishes to use with Perl. The XS interface is combined with the library to create a new library which can be linked to Perl. An **XSUB** is a function in the XS language and is the core component of the Perl application interface.

The XS compiler is called **xsubpp**. This compiler will embed the constructs necessary to let an XSUB, which is really a C function in disguise, manipulate Perl values and creates the glue necessary to let Perl access the XSUB. The compiler uses **typemaps** to determine how to map C function parameters and variables to Perl values. The default typemap handles many common C types. A supplement typemap must be created to handle special structures and types for the library being linked.

See the *perlxstut* manpage for a tutorial on the whole extension creation process.

Note: For many extensions, Dave Beazley's SWIG system provides a significantly more convenient mechanism for creating the XS glue code. See the section on */www.cs.utah.edu/~beazley/SWIG* in the *http:* manpage for more information.

### On The Road

Many of the examples which follow will concentrate on creating an interface between Perl and the ONC+ RPC bind library functions. The *rpcb_gettime()* function is used to demonstrate many features of the XS language. This function has two parameters; the first is an input parameter and the second is an output parameter. The function also returns a status value.

```
bool_t rpcb_gettime(const char *host, time_t *timep);
```

From C this function will be called with the following statements.

```
#include <rpc/rpc.h>
bool_t status;
time_t timep;
status = rpcb_gettime( "localhost", &timep );
```

If an XSUB is created to offer a direct translation between this function and Perl, then this XSUB will be used from Perl with the following code. The `$status` and `$timep` variables will contain the output of the function.

```
use RPC;
$status = rpcb_gettime( "localhost", $timep );
```

The following XS file shows an XS subroutine, or XSUB, which demonstrates one possible interface to the *rpcb_gettime()* function. This XSUB represents a direct translation between C and Perl and so preserves the interface even from Perl. This XSUB will be invoked from Perl with the usage shown above. Note that the first three #include statements, for `EXTERN.h`, `perl.h`, and `XSUB.h`, will always be present at the beginning of an XS file. This approach and others will be expanded later in this document.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include <rpc/rpc.h>

MODULE = RPC  PACKAGE = RPC
```

```
bool_t
rpcb_gettime(host,timep)
     char *host
     time_t &timep
     OUTPUT:
     timep
```

Any extension to Perl, including those containing XSUBs, should have a Perl module to serve as the bootstrap which pulls the extension into Perl. This module will export the extension's functions and variables to the Perl program and will cause the extension's XSUBs to be linked into Perl. The following module will be used for most of the examples in this document and should be used from Perl with the `use` command as shown earlier. Perl modules are explained in more detail later in this document.

```
package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw( rpcb_gettime );

bootstrap RPC;
1;
```

Throughout this document a variety of interfaces to the *rpcb_gettime()* XSUB will be explored. The XSUBs will take their parameters in different orders or will take different numbers of parameters. In each case the XSUB is an abstraction between Perl and the real C *rpcb_gettime()* function, and the XSUB must always ensure that the real *rpcb_gettime()* function is called with the correct parameters. This abstraction will allow the programmer to create a more Perl-like interface to the C function.

### The Anatomy of an XSUB

The following XSUB allows a Perl program to access a C library function called *sin()*. The XSUB will imitate the C function which takes a single argument and returns a single value.

```
double
sin(x)
  double x
```

When using C pointers the indirection operator `*` should be considered part of the type and the address operator `&` should be considered part of the variable, as is demonstrated in the *rpcb_gettime()* function above. See the section on typemaps for more about handling qualifiers and unary operators in C types.

The function name and the return type must be placed on separate lines.

```
INCORRECT                        CORRECT

double sin(x)                    double
  double x                       sin(x)
                                   double x
```

The function body may be indented or left-adjusted. The following example shows a function with its body left-adjusted. Most examples in this document will indent the body.

```
CORRECT
```

```
double
sin(x)
double x
```

### The Argument Stack

The argument stack is used to store the values which are sent as parameters to the XSUB and to store the XSUB's return value. In reality all Perl functions keep their values on this stack at the same time, each limited to its own range of positions on the stack. In this document the first position on that stack which belongs to the active function will be referred to as position 0 for that function.

XSUBs refer to their stack arguments with the macro **ST(x)**, where *x* refers to a position in this XSUB's part of the stack. Position 0 for that function would be known to the XSUB as *ST*(0). The XSUB's incoming parameters and outgoing return values always begin at *ST*(0). For many simple cases the **xsubpp** compiler will generate the code necessary to handle the argument stack by embedding code fragments found in the typemaps. In more complex cases the programmer must supply the code.

### The RETVAL Variable

The RETVAL variable is a magic variable which always matches the return type of the C library function. The **xsubpp** compiler will supply this variable in each XSUB and by default will use it to hold the return value of the C library function being called. In simple cases the value of RETVAL will be placed in *ST*(0) of the argument stack where it can be received by Perl as the return value of the XSUB.

If the XSUB has a return type of `void` then the compiler will not supply a RETVAL variable for that function. When using the PPCODE: directive the RETVAL variable is not needed, unless used explicitly.

If PPCODE: directive is not used, `void` return value should be used only for subroutines which do not return a value, *even if* CODE: directive is used which sets *ST*(0) explicitly.

Older versions of this document recommended to use `void` return value in such cases. It was discovered that this could lead to segfaults in cases when XSUB was *truely* `void`. This practice is now deprecated, and may be not supported at some future version. Use the return value `SV *` in such cases. (Currently `xsubpp` contains some heuristic code which tries to disambiguate between "truely-void" and "old-practice-declared-as-void" functions. Hence your code is at mercy of this heuristics unless you use `SV *` as return value.)

### The MODULE Keyword

The MODULE keyword is used to start the XS code and to specify the package of the functions which are being defined. All text preceding the first MODULE keyword is considered C code and is passed through to the output untouched. Every XS module will have a bootstrap function which is used to hook the XSUBs into Perl. The package name of this bootstrap function will match the value of the last MODULE statement in the XS source files. The value of MODULE should always remain constant within the same XS file, though this is not required.

The following example will start the XS code and will place all functions in a package named RPC.

```
MODULE = RPC
```

### The PACKAGE Keyword

When functions within an XS source file must be separated into packages the PACKAGE keyword should be used. This keyword is used with the MODULE keyword and must follow immediately after it when used.

```
MODULE = RPC  PACKAGE = RPC

[ XS code in package RPC ]
```

```
MODULE = RPC  PACKAGE = RPCB

[ XS code in package RPCB ]

MODULE = RPC  PACKAGE = RPC

[ XS code in package RPC ]
```

Although this keyword is optional and in some cases provides redundant information it should always be used. This keyword will ensure that the XSUBs appear in the desired package.

### The PREFIX Keyword

The PREFIX keyword designates prefixes which should be removed from the Perl function names. If the C function is `rpcb_gettime()` and the PREFIX value is `rpcb_` then Perl will see this function as `gettime()`.

This keyword should follow the PACKAGE keyword when used. If PACKAGE is not used then PREFIX should follow the MODULE keyword.

```
MODULE = RPC  PREFIX = rpc_

MODULE = RPC  PACKAGE = RPCB  PREFIX = rpcb_
```

### The OUTPUT: Keyword

The OUTPUT: keyword indicates that certain function parameters should be updated (new values made visible to Perl) when the XSUB terminates or that certain values should be returned to the calling Perl function. For simple functions, such as the *sin()* function above, the RETVAL variable is automatically designated as an output value. In more complex functions the **xsubpp** compiler will need help to determine which variables are output variables.

This keyword will normally be used to complement the CODE: keyword. The RETVAL variable is not recognized as an output variable when the CODE: keyword is present. The OUTPUT: keyword is used in this situation to tell the compiler that RETVAL really is an output variable.

The OUTPUT: keyword can also be used to indicate that function parameters are output variables. This may be necessary when a parameter has been modified within the function and the programmer would like the update to be seen by Perl.

```
bool_t
rpcb_gettime(host,timep)
     char *host
     time_t &timep
     OUTPUT:
     timep
```

The OUTPUT: keyword will also allow an output parameter to be mapped to a matching piece of code rather than to a typemap.

```
bool_t
rpcb_gettime(host,timep)
     char *host
     time_t &timep
     OUTPUT:
     timep sv_setnv(ST(1), (double)timep);
```

**xsubpp** emits an automatic `SvSETMAGIC()` for all parameters in the OUTPUT section of the XSUB, except RETVAL. This is the usually desired behavior, as it takes care of properly invoking 'set' magic on output parameters (needed for hash or array element parameters that must be created if they didn't exist). If for some reason, this behavior is not desired, the OUTPUT section may contain a `SETMAGIC: DISABLE` line to disable it for the remainder of the parameters in the OUTPUT section. Likewise, `SETMAGIC: ENABLE` can be used to reenable it for the remainder of the OUTPUT section. See the *perlguts* manpage for more details about 'set' magic.

### The CODE: Keyword

This keyword is used in more complicated XSUBs which require special handling for the C function. The RETVAL variable is available but will not be returned unless it is specified under the OUTPUT: keyword.

The following XSUB is for a C function which requires special handling of its parameters. The Perl usage is given first.

```
$status = rpcb_gettime( "localhost", $timep );
```

The XSUB follows.

```
bool_t
rpcb_gettime(host,timep)
     char *host
     time_t timep
     CODE:
          RETVAL = rpcb_gettime( host, &timep );
     OUTPUT:
     timep
     RETVAL
```

### The INIT: Keyword

The INIT: keyword allows initialization to be inserted into the XSUB before the compiler generates the call to the C function. Unlike the CODE: keyword above, this keyword does not affect the way the compiler handles RETVAL.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t &timep
      INIT:
      printf("# Host is %s\n", host );
      OUTPUT:
      timep
```

### The NO_INIT Keyword

The NO_INIT keyword is used to indicate that a function parameter is being used only as an output value. The **xsubpp** compiler will normally generate code to read the values of all function parameters from the argument stack and assign them to C variables upon entry to the function. NO_INIT will tell the compiler that some parameters will be used for output rather than for input and that they will be handled before the function terminates.

The following example shows a variation of the *rpcb_gettime()* function. This function uses the timep variable only as an output variable and does not care about its initial contents.

```
bool_t
rpcb_gettime(host,timep)
     char *host
     time_t &timep = NO_INIT
     OUTPUT:
     timep
```

**Initializing Function Parameters**

Function parameters are normally initialized with their values from the argument stack. The typemaps contain the code segments which are used to transfer the Perl values to the C parameters. The programmer, however, is allowed to override the typemaps and supply alternate (or additional) initialization code.

The following code demonstrates how to supply initialization code for function parameters. The initialization code is eval'd within double quotes by the compiler before it is added to the output so anything which should be interpreted literally [mainly `$`, `@`, or `\\`] must be protected with backslashes. The variables `$var`, `$arg`, and `$type` can be used as in typemaps.

```
bool_t
rpcb_gettime(host,timep)
     char *host = (char *)SvPV($arg,PL_na);
     time_t &timep = 0;
     OUTPUT:
     timep
```

This should not be used to supply default values for parameters. One would normally use this when a function parameter must be processed by another library function before it can be used. Default parameters are covered in the next section.

If the initialization begins with =, then it is output on the same line where the input variable is declared. If the initialization begins with `;` or +, then it is output after all of the input variables have been declared. The = and `;` cases replace the initialization normally supplied from the typemap. For the + case, the initialization from the typemap will preceed the initialization code included after the +. A global variable, `%v`, is available for the truely rare case where information from one initialization is needed in another initialization.

```
bool_t
rpcb_gettime(host,timep)
     time_t &timep ; /*\$v{time}=@{[$v{time}=$arg]}*/
     char *host + SvOK($v{time}) ? SvPV($arg,PL_na) : NULL;
     OUTPUT:
     timep
```

**Default Parameter Values**

Default values can be specified for function parameters by placing an assignment statement in the parameter list. The default value may be a number or a string. Defaults should always be used on the right-most parameters only.

To allow the XSUB for *rpcb_gettime()* to have a default host value the parameters to the XSUB could be rearranged. The XSUB will then call the real *rpcb_gettime()* function with the parameters in the correct order. Perl will call this XSUB with either of the following statements.

```
$status = rpcb_gettime( $timep, $host );

$status = rpcb_gettime( $timep );
```

The XSUB will look like the code which follows. A CODE: block is used to call the real *rpcb_gettime()* function with the parameters in the correct order for that function.

```
bool_t
rpcb_gettime(timep,host="localhost")
     char *host
     time_t timep = NO_INIT
     CODE:
          RETVAL = rpcb_gettime( host, &timep );
     OUTPUT:
     timep
     RETVAL
```

### The PREINIT: Keyword

The PREINIT: keyword allows extra variables to be declared before the typemaps are expanded. If a variable is declared in a CODE: block then that variable will follow any typemap code. This may result in a C syntax error. To force the variable to be declared before the typemap code, place it into a PREINIT: block. The PREINIT: keyword may be used one or more times within an XSUB.

The following examples are equivalent, but if the code is using complex typemaps then the first example is safer.

```
bool_t
rpcb_gettime(timep)
     time_t timep = NO_INIT
     PREINIT:
     char *host = "localhost";
     CODE:
     RETVAL = rpcb_gettime( host, &timep );
     OUTPUT:
     timep
     RETVAL
```

A correct, but error-prone example.

```
bool_t
rpcb_gettime(timep)
     time_t timep = NO_INIT
     CODE:
     char *host = "localhost";
     RETVAL = rpcb_gettime( host, &timep );
     OUTPUT:
     timep
     RETVAL
```

### The SCOPE: Keyword

The SCOPE: keyword allows scoping to be enabled for a particular XSUB. If enabled, the XSUB will invoke ENTER and LEAVE automatically.

To support potentially complex type mappings, if a typemap entry used by this XSUB contains a comment like `/*scope*/` then scoping will automatically be enabled for that XSUB.

To enable scoping:

```
SCOPE: ENABLE
```

To disable scoping:

```
SCOPE: DISABLE
```

**The INPUT: Keyword**

The XSUB's parameters are usually evaluated immediately after entering the XSUB. The INPUT: keyword can be used to force those parameters to be evaluated a little later. The INPUT: keyword can be used multiple times within an XSUB and can be used to list one or more input variables. This keyword is used with the PREINIT: keyword.

The following example shows how the input parameter `timep` can be evaluated late, after a PREINIT.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      PREINIT:
      time_t tt;
      INPUT:
      time_t timep
      CODE:
           RETVAL = rpcb_gettime( host, &tt );
           timep = tt;
      OUTPUT:
      timep
      RETVAL
```

The next example shows each input parameter evaluated late.

```
bool_t
rpcb_gettime(host,timep)
      PREINIT:
      time_t tt;
      INPUT:
      char *host
      PREINIT:
      char *h;
      INPUT:
      time_t timep
      CODE:
           h = host;
           RETVAL = rpcb_gettime( h, &tt );
           timep = tt;
      OUTPUT:
      timep
      RETVAL
```

**Variable-length Parameter Lists**

XSUBs can have variable-length parameter lists by specifying an ellipsis `(...)` in the parameter list. This use of the ellipsis is similar to that found in ANSI C. The programmer is able to determine the number of arguments passed to the XSUB by examining the `items` variable which the **xsubpp** compiler supplies for all XSUBs. By using this mechanism one can create an XSUB which accepts a list of parameters of unknown length.

The *host* parameter for the *rpcb_gettime()* XSUB can be optional so the ellipsis can be used to indicate that the XSUB will take a variable number of parameters. Perl should be able to call this XSUB with either of the following statements.

```
$status = rpcb_gettime( $timep, $host );

$status = rpcb_gettime( $timep );
```

The XS code, with ellipsis, follows.

```
bool_t
rpcb_gettime(timep, ...)
     time_t timep = NO_INIT
     PREINIT:
     char *host = "localhost";
     CODE:
             if( items > 1 )
                  host = (char *)SvPV(ST(1), PL_na);
             RETVAL = rpcb_gettime( host, &timep );
     OUTPUT:
     timep
     RETVAL
```

### The C_ARGS: Keyword

The C_ARGS: keyword allows creating of XSUBS which have different calling sequence from Perl than from C, without a need to write CODE: or CPPCODE: section. The contents of the C_ARGS: paragraph is put as the argument to the called C function without any change.

For example, suppose that C function is declared as

```
symbolic nth_derivative(int n, symbolic function, int flags);
```

and that the default flags are kept in a global C variable `default_flags`. Suppose that you want to create an interface which is called as

```
$second_deriv = $function->nth_derivative(2);
```

To do this, declare the XSUB as

```
symbolic
nth_derivative(function, n)
    symbolic        function
    int             n
C_ARGS:
    n, function, default_flags
```

### The PPCODE: Keyword

The PPCODE: keyword is an alternate form of the CODE: keyword and is used to tell the **xsubpp** compiler that the programmer is supplying the code to control the argument stack for the XSUBs return values. Occasionally one will want an XSUB to return a list of values rather than a single value. In these cases one must use PPCODE: and then explicitly push the list of values on the stack. The PPCODE: and CODE: keywords are not used together within the same XSUB.

The following XSUB will call the C *rpcb_gettime()* function and will return its two output values, timep and status, to Perl as a single list.

```
void
rpcb_gettime(host)
     char *host
     PREINIT:
     time_t  timep;
     bool_t  status;
     PPCODE:
     status = rpcb_gettime( host, &timep );
     EXTEND(SP, 2);
     PUSHs(sv_2mortal(newSViv(status)));
     PUSHs(sv_2mortal(newSViv(timep)));
```

Notice that the programmer must supply the C code necessary to have the real *rpcb_gettime()* function called and to have the return values properly placed on the argument stack.

The `void` return type for this function tells the **xsubpp** compiler that the RETVAL variable is not needed or used and that it should not be created. In most scenarios the void return type should be used with the PPCODE: directive.

The *EXTEND()* macro is used to make room on the argument stack for 2 return values. The PPCODE: directive causes the **xsubpp** compiler to create a stack pointer available as `SP`, and it is this pointer which is being used in the *EXTEND()* macro. The values are then pushed onto the stack with the *PUSHs()* macro.

Now the *rpcb_gettime()* function can be used from Perl with the following statement.

```
($status, $timep) = rpcb_gettime("localhost");
```

When handling output parameters with a PPCODE section, be sure to handle 'set' magic properly. See the *perlguts* manpage for details about 'set' magic.

### Returning Undef And Empty Lists

Occasionally the programmer will want to return simply `undef` or an empty list if a function fails rather than a separate status value. The *rpcb_gettime()* function offers just this situation. If the function succeeds we would like to have it return the time and if it fails we would like to have undef returned. In the following Perl code the value of `$timep` will either be undef or it will be a valid time.

```
$timep = rpcb_gettime( "localhost" );
```

The following XSUB uses the `SV *` return type as a mnemonic only, and uses a CODE: block to indicate to the compiler that the programmer has supplied all the necessary code. The *sv_newmortal()* call will initialize the return value to undef, making that the default return value.

```
SV *
rpcb_gettime(host)
     char *  host
     PREINIT:
     time_t  timep;
     bool_t x;
     CODE:
     ST(0) = sv_newmortal();
     if( rpcb_gettime( host, &timep ) )
          sv_setnv( ST(0), (double)timep);
```

The next example demonstrates how one would place an explicit undef in the return value, should the need arise.

```
SV *
rpcb_gettime(host)
     char *  host
     PREINIT:
     time_t  timep;
     bool_t x;
     CODE:
     ST(0) = sv_newmortal();
     if( rpcb_gettime( host, &timep ) ){
          sv_setnv( ST(0), (double)timep);
     }
     else{
          ST(0) = &PL_sv_undef;
     }
```

To return an empty list one must use a PPCODE: block and then not push return values on the stack.

```
void
rpcb_gettime(host)
     char *host
     PREINIT:
     time_t  timep;
     PPCODE:
     if( rpcb_gettime( host, &timep ) )
          PUSHs(sv_2mortal(newSViv(timep)));
     else{
     /* Nothing pushed on stack, so an empty */
     /* list is implicitly returned. */
     }
```

Some people may be inclined to include an explicit `return` in the above XSUB, rather than letting control fall through to the end. In those situations `XSRETURN_EMPTY` should be used, instead. This will ensure that the XSUB stack is properly adjusted. Consult the section on *API LISTING* in the *perlguts* manpage for other `XSRETURN` macros.

### The REQUIRE: Keyword

The REQUIRE: keyword is used to indicate the minimum version of the **xsubpp** compiler needed to compile the XS module. An XS module which contains the following statement will compile with only **xsubpp** version 1.922 or greater:

```
REQUIRE: 1.922
```

### The CLEANUP: Keyword

This keyword can be used when an XSUB requires special cleanup procedures before it terminates. When the CLEANUP: keyword is used it must follow any CODE:, PPCODE:, or OUTPUT: blocks which are present in the XSUB. The code specified for the cleanup block will be added as the last statements in the XSUB.

### The BOOT: Keyword

The BOOT: keyword is used to add code to the extension's bootstrap function. The bootstrap function is generated by the **xsubpp** compiler and normally holds the statements necessary to register any XSUBs with Perl. With the BOOT: keyword the programmer can tell the compiler to add extra statements to the bootstrap function.

This keyword may be used any time after the first MODULE keyword and should appear on a line by itself.

The first blank line after the keyword will terminate the code block.

```
BOOT:
# The following message will be printed when the
# bootstrap function executes.
printf("Hello from the bootstrap!\n");
```

### The VERSIONCHECK: Keyword

The VERSIONCHECK: keyword corresponds to **xsubpp**'s `-versioncheck` and `-noversioncheck` options. This keyword overrides the command line options. Version checking is enabled by default. When version checking is enabled the XS module will attempt to verify that its version matches the version of the PM module.

To enable version checking:

```
VERSIONCHECK: ENABLE
```

To disable version checking:

```
VERSIONCHECK: DISABLE
```

### The PROTOTYPES: Keyword

The PROTOTYPES: keyword corresponds to **xsubpp**'s `-prototypes` and `-noprototypes` options. This keyword overrides the command line options. Prototypes are enabled by default. When prototypes are enabled XSUBs will be given Perl prototypes. This keyword may be used multiple times in an XS module to enable and disable prototypes for different parts of the module.

To enable prototypes:

```
PROTOTYPES: ENABLE
```

To disable prototypes:

```
PROTOTYPES: DISABLE
```

### The PROTOTYPE: Keyword

This keyword is similar to the PROTOTYPES: keyword above but can be used to force **xsubpp** to use a specific prototype for the XSUB. This keyword overrides all other prototype options and keywords but affects only the current XSUB. Consult the `Prototypes` entry in the *perlsub* manpage for information about Perl prototypes.

```
bool_t
rpcb_gettime(timep, ...)
      time_t timep = NO_INIT
      PROTOTYPE: $;$
      PREINIT:
      char *host = "localhost";
      CODE:
              if( items > 1 )
                   host = (char *)SvPV(ST(1), PL_na);
              RETVAL = rpcb_gettime( host, &timep );
      OUTPUT:
      timep
      RETVAL
```

### The ALIAS: Keyword

The ALIAS: keyword allows an XSUB to have two or more unique Perl names and to know which of those names was used when it was invoked. The Perl names may be fully-qualified with package names. Each alias is given an index. The compiler will setup a variable called `ix` which contain the index of the alias which was used. When the XSUB is called with its declared name `ix` will be 0.

The following example will create aliases `FOO::gettime()` and `BAR::getit()` for this function.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t &timep
      ALIAS:
        FOO::gettime = 1
        BAR::getit = 2
      INIT:
      printf("# ix = %d\n", ix );
      OUTPUT:
      timep
```

### The INTERFACE: Keyword

This keyword declares the current XSUB as a keeper of the given calling signature. If some text follows this keyword, it is considered as a list of functions which have this signature, and should be attached to XSUBs.

Say, if you have 4 functions *multiply()*, *divide()*, *add()*, *subtract()* all having the signature

```
symbolic f(symbolic, symbolic);
```

you code them all by using XSUB

```
symbolic
interface_s_ss(arg1, arg2)
    symbolic        arg1
    symbolic        arg2
INTERFACE:
    multiply divide
    add subtract
```

The advantage of this approach comparing to ALIAS: keyword is that one can attach an extra function *remainder()* at runtime by using

CV *mycv = *newXSproto* ("Symbolic::remainder”, XS_Symbolic_interface_s_ss, _ _FILE_ _, “$$");
*XSINTERFACE_FUNC_SET* (mycv, remainder);

(This example supposes that there was no INTERFACE_MACRO: section, otherwise one needs to use something else instead of `XSINTERFACE_FUNC_SET`.)

### The INTERFACE_MACRO: Keyword

This keyword allows one to define an INTERFACE using a different way to extract a function pointer from an XSUB. The text which follows this keyword should give the name of macros which would extract/set a function pointer. The extractor macro is given return type, `CV*`, and `XSANY.any_dptr` for this `CV*`. The setter macro is given cv, and the function pointer.

The default value is `XSINTERFACE_FUNC` and `XSINTERFACE_FUNC_SET`. An INTERFACE keyword with an empty list of functions can be omitted if INTERFACE_MACRO keyword is used.

Suppose that in the previous example functions pointers for *multiply()*, *divide()*, *add()*, *subtract()* are kept in a global C array `fp[]` with offsets being `multiply_off`, `divide_off`, `add_off`, `subtract_off`. Then one can use

```
#define XSINTERFACE_FUNC_BYOFFSET(ret,cv,f) \
    ((XSINTERFACE_CVT(ret,))fp[CvXSUBANY(cv).any_i32])
#define XSINTERFACE_FUNC_BYOFFSET_set(cv,f) \
    CvXSUBANY(cv).any_i32 = CAT2( f, _off )
```

in C section,

```
symbolic
interface_s_ss(arg1, arg2)
    symbolic        arg1
    symbolic        arg2
INTERFACE_MACRO:
    XSINTERFACE_FUNC_BYOFFSET
    XSINTERFACE_FUNC_BYOFFSET_set
INTERFACE:
    multiply divide
    add subtract
```

in XSUB section.

### The INCLUDE: Keyword

This keyword can be used to pull other files into the XS module. The other files may have XS code. INCLUDE: can also be used to run a command to generate the XS code to be pulled into the module.

The file *Rpcb1.xsh* contains our `rpcb_gettime()` function:

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t &timep
      OUTPUT:
      timep
```

The XS module can use INCLUDE: to pull that file into it.

```
INCLUDE: Rpcb1.xsh
```

If the parameters to the INCLUDE: keyword are followed by a pipe (|) then the compiler will interpret the parameters as a command.

```
INCLUDE: cat Rpcb1.xsh |
```

### The CASE: Keyword

The CASE: keyword allows an XSUB to have multiple distinct parts with each part acting as a virtual XSUB. CASE: is greedy and if it is used then all other XS keywords must be contained within a CASE:. This means nothing may precede the first CASE: in the XSUB and anything following the last CASE: is included in that case.

A CASE: might switch via a parameter of the XSUB, via the `ix` ALIAS: variable (see the section on *The ALIAS: Keyword*), or maybe via the `items` variable (see the section on *Variable-length Parameter Lists*). The last CASE: becomes the **default** case if it is not associated with a conditional. The following example shows CASE switched via `ix` with a function `rpcb_gettime()` having an alias `x_gettime()`. When the function is called as `rpcb_gettime()` its parameters are the usual `(char *host, time_t` `*timep)`, but when the function is called as `x_gettime()` its parameters are reversed, `(time_t *timep, char *host)`.

```
long
rpcb_gettime(a,b)
  CASE: ix == 1
      ALIAS:
      x_gettime = 1
      INPUT:
      # 'a' is timep, 'b' is host
      char *b
      time_t a = NO_INIT
      CODE:
           RETVAL = rpcb_gettime( b, &a );
      OUTPUT:
      a
      RETVAL
  CASE:
      # 'a' is host, 'b' is timep
      char *a
      time_t &b = NO_INIT
      OUTPUT:
      b
      RETVAL
```

That function can be called with either of the following statements. Note the different argument lists.

```
$status = rpcb_gettime( $host, $timep );

$status = x_gettime( $timep, $host );
```

### The & Unary Operator

The & unary operator is used to tell the compiler that it should dereference the object when it calls the C function. This is used when a CODE: block is not used and the object is a not a pointer type (the object is an `int` or `long` but not a `int*` or `long*`).

The following XSUB will generate incorrect C code. The xsubpp compiler will turn this into code which calls `rpcb_gettime()` with parameters `(char *host, time_t timep)`, but the real `rpcb_gettime()` wants the `timep` parameter to be of type `time_t*` rather than `time_t`.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t timep
      OUTPUT:
      timep
```

That problem is corrected by using the `&` operator. The xsubpp compiler will now turn this into code which calls `rpcb_gettime()` correctly with parameters `(char *host, time_t *timep)`. It does this by carrying the `&` through, so the function call looks like `rpcb_gettime(host, &timep)`.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t &timep
      OUTPUT:
      timep
```

### Inserting Comments and C Preprocessor Directives

C preprocessor directives are allowed within BOOT:, PREINIT: INIT:, CODE:, PPCODE:, and CLEANUP: blocks, as well as outside the functions. Comments are allowed anywhere after the MODULE keyword. The compiler will pass the preprocessor directives through untouched and will remove the commented lines.

Comments can be added to XSUBs by placing a `#` as the first non-whitespace of a line. Care should be taken to avoid making the comment look like a C preprocessor directive, lest it be interpreted as such. The simplest way to prevent this is to put whitespace in front of the `#`.

If you use preprocessor directives to choose one of two versions of a function, use

```
#if ... version1
#else /* ... version2  */
#endif
```

and not

```
#if ... version1
#endif
#if ... version2
#endif
```

because otherwise xsubpp will believe that you made a duplicate definition of the function. Also, put a blank line before the #else/#endif so it will not be seen as part of the function body.

### Using XS With C++

If a function is defined as a C++ method then it will assume its first argument is an object pointer. The object pointer will be stored in a variable called THIS. The object should have been created by C++ with the *new()* function and should be blessed by Perl with the *sv_setref_pv()* macro. The blessing of the object by Perl can be handled by a typemap. An example typemap is shown at the end of this section.

If the method is defined as static it will call the C++ function using the *class::method()* syntax. If the method is not static the function will be called using the THIS−>*method()* syntax.

The next examples will use the following C++ class.

```
 class color {
      public:
      color();
      ~color();
      int blue();
      void set_blue( int );

      private:
      int c_blue;
 };
```

The XSUBs for the *blue()* and *set_blue()* methods are defined with the class name but the parameter for the object (THIS, or “self") is implicit and is not listed.

```
int
color::blue()

void
color::set_blue( val )
     int val
```

Both functions will expect an object as the first parameter. The xsubpp compiler will call that object `THIS` and will use it to call the specified method. So in the C++ code the *blue()* and *set_blue()* methods will be called in the following manner.

```
RETVAL = THIS->blue();

THIS->set_blue( val );
```

If the function's name is **DESTROY** then the C++ `delete` function will be called and `THIS` will be given as its parameter.

```
void
color::DESTROY()
```

The C++ code will call `delete`.

```
delete THIS;
```

If the function's name is **new** then the C++ `new` function will be called to create a dynamic C++ object. The XSUB will expect the class name, which will be kept in a variable called `CLASS`, to be given as the first argument.

```
color *
color::new()
```

The C++ code will call `new`.

```
   RETVAL = new color();
```

The following is an example of a typemap that could be used for this C++ example.

```
TYPEMAP
color *             O_OBJECT

OUTPUT
# The Perl object is blessed into 'CLASS', which should be a
# char* having the name of the package for the blessing.
O_OBJECT
    sv_setref_pv( $arg, CLASS, (void*)$var );

INPUT
O_OBJECT
    if( sv_isobject($arg) && (SvTYPE(SvRV($arg)) == SVt_PVMG) )
            $var = ($type)SvIV((SV*)SvRV( $arg ));
    else{
            warn( \"${Package}::$func_name() -- $var is not a blessed SV referen
            XSRETURN_UNDEF;
    }
```

### Interface Strategy

When designing an interface between Perl and a C library a straight translation from C to XS is often sufficient. The interface will often be very C−like and occasionally nonintuitive, especially when the C function modifies one of its parameters. In cases where the programmer wishes to create a more Perl-like interface the following strategy may help to identify the more critical parts of the interface.

Identify the C functions which modify their parameters. The XSUBs for these functions may be able to return lists to Perl, or may be candidates to return undef or an empty list in case of failure.

Identify which values are used by only the C and XSUB functions themselves. If Perl does not need to access the contents of the value then it may not be necessary to provide a translation for that value from C to Perl.

Identify the pointers in the C function parameter lists and return values. Some pointers can be handled in XS with the & unary operator on the variable name while others will require the use of the * operator on the type name. In general it is easier to work with the & operator.

Identify the structures used by the C functions. In many cases it may be helpful to use the T_PTROBJ typemap for these structures so they can be manipulated by Perl as blessed objects.

### Perl Objects And C Structures

When dealing with C structures one should select either **T_PTROBJ** or **T_PTRREF** for the XS type. Both types are designed to handle pointers to complex objects. The T_PTRREF type will allow the Perl object to be unblessed while the T_PTROBJ type requires that the object be blessed. By using T_PTROBJ one can achieve a form of type-checking because the XSUB will attempt to verify that the Perl object is of the expected type.

The following XS code shows the *getnetconfigent()* function which is used with ONC+ TIRPC. The *getnetconfigent()* function will return a pointer to a C structure and has the C prototype shown below. The example will demonstrate how the C pointer will become a Perl reference. Perl will consider this reference to be a pointer to a blessed object and will attempt to call a destructor for the object. A destructor will be provided in the XS source to free the memory used by *getnetconfigent()*. Destructors in XS can be created by specifying an XSUB function whose name ends with the word **DESTROY**. XS destructors can be used to free memory which may have been malloc'd by another XSUB.

```
struct netconfig *getnetconfigent(const char *netid);
```

A `typedef` will be created for `struct netconfig`. The Perl object will be blessed in a class matching the name of the C type, with the tag `Ptr` appended, and the name should not have embedded spaces if it will be a Perl package name. The destructor will be placed in a class corresponding to the class of the object and the PREFIX keyword will be used to trim the name to the word DESTROY as Perl will expect.

```
typedef struct netconfig Netconfig;

MODULE = RPC  PACKAGE = RPC

Netconfig *
getnetconfigent(netid)
     char *netid

MODULE = RPC  PACKAGE = NetconfigPtr  PREFIX = rpcb_
```

```
void
rpcb_DESTROY(netconf)
     Netconfig *netconf
     CODE:
     printf("Now in NetconfigPtr::DESTROY\n");
     free( netconf );
```

This example requires the following typemap entry. Consult the typemap section for more information about adding new typemaps for an extension.

```
TYPEMAP
Netconfig *  T_PTROBJ
```

This example will be used with the following Perl statements.

```
use RPC;
$netconf = getnetconfigent("udp");
```

When Perl destroys the object referenced by `$netconf` it will send the object to the supplied XSUB DESTROY function. Perl cannot determine, and does not care, that this object is a C struct and not a Perl object. In this sense, there is no difference between the object created by the *getnetconfigent()* XSUB and an object created by a normal Perl subroutine.

### The Typemap

The typemap is a collection of code fragments which are used by the **xsubpp** compiler to map C function parameters and values to Perl values. The typemap file may consist of three sections labeled `TYPEMAP`, `INPUT`, and `OUTPUT`. The INPUT section tells the compiler how to translate Perl values into variables of certain C types. The OUTPUT section tells the compiler how to translate the values from certain C types into values Perl can understand. The TYPEMAP section tells the compiler which of the INPUT and OUTPUT code fragments should be used to map a given C type to a Perl value. Each of the sections of the typemap must be preceded by one of the TYPEMAP, INPUT, or OUTPUT keywords.

The default typemap in the `ext` directory of the Perl source contains many useful types which can be used by Perl extensions. Some extensions define additional typemaps which they keep in their own directory. These additional typemaps may reference INPUT and OUTPUT maps in the main typemap. The **xsubpp** compiler will allow the extension's own typemap to override any mappings which are in the default typemap.

Most extensions which require a custom typemap will need only the TYPEMAP section of the typemap file. The custom typemap used in the *getnetconfigent()* example shown earlier demonstrates what may be the typical use of extension typemaps. That typemap is used to equate a C structure with the T_PTROBJ typemap. The typemap used by *getnetconfigent()* is shown here. Note that the C type is separated from the XS type with a tab and that the C unary operator `*` is considered to be a part of the C type name.

```
TYPEMAP
Netconfig *<tab>T_PTROBJ
```

Here's a more complicated example: suppose that you wanted `struct netconfig` to be blessed into the class `Net::Config`. One way to do this is to use underscores (_) to separate package names, as follows:

```
typedef struct netconfig * Net_Config;
```

And then provide a typemap entry `T_PTROBJ_SPECIAL` that maps underscores to double-colons (::), and declare `Net_Config` to be of that type:

```
TYPEMAP
Net_Config      T_PTROBJ_SPECIAL

INPUT
T_PTROBJ_SPECIAL
        if (sv_derived_from($arg, \"${(my $ntt=$ntype)=~s/_/::/g;\$ntt}\"))
                IV tmp = SvIV((SV*)SvRV($arg));
        $var = ($type) tmp;
        }
        else
                croak(\"$var is not of type ${(my $ntt=$ntype)=~s/_/::/g;\$n

OUTPUT
T_PTROBJ_SPECIAL
        sv_setref_pv($arg, \"${(my $ntt=$ntype)=~s/_/::/g;\$ntt}\",
        (void*)$var);
```

The INPUT and OUTPUT sections substitute underscores for double-colons on the fly, giving the desired effect. This example demonstrates some of the power and versatility of the typemap facility.

**EXAMPLES**

File `RPC.xs`: Interface to some ONC+ RPC bind library functions.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include <rpc/rpc.h>

typedef struct netconfig Netconfig;

MODULE = RPC  PACKAGE = RPC

SV *
rpcb_gettime(host="localhost")
     char *host
     PREINIT:
     time_t  timep;
     CODE:
     ST(0) = sv_newmortal();
     if( rpcb_gettime( host, &timep ) )
          sv_setnv( ST(0), (double)timep );

Netconfig *
getnetconfigent(netid="udp")
     char *netid

MODULE = RPC  PACKAGE = NetconfigPtr  PREFIX = rpcb_

void
rpcb_DESTROY(netconf)
     Netconfig *netconf
     CODE:
     printf("NetconfigPtr::DESTROY\n");
     free( netconf );
```

File `typemap`: Custom typemap for RPC.xs.

```
TYPEMAP
Netconfig *  T_PTROBJ
```

File `RPC.pm`: Perl module for the RPC extension.

```
package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw(rpcb_gettime getnetconfigent);

bootstrap RPC;
1;
```

File `rpctest.pl`: Perl test program for the RPC extension.

```
use RPC;

$netconf = getnetconfigent();
$a = rpcb_gettime();
print "time = $a\n";
print "netconf = $netconf\n";

$netconf = getnetconfigent("tcp");
$a = rpcb_gettime("poplar");
print "time = $a\n";
print "netconf = $netconf\n";
```

**XS VERSION**

This document covers features supported by `xsubpp` 1.935.

**AUTHOR**

Dean Roehrich <*roehrich@cray.com*> Jul 8, 1996

## NAME

perlXStut − Tutorial for XSUBs

## DESCRIPTION

This tutorial will educate the reader on the steps involved in creating a Perl extension. The reader is assumed to have access to the *perlguts* manpage and the *perlxs* manpage.

This tutorial starts with very simple examples and becomes more complex, with each new example adding new features. Certain concepts may not be completely explained until later in the tutorial to ease the reader slowly into building extensions.

### VERSION CAVEAT

This tutorial tries hard to keep up with the latest development versions of Perl. This often means that it is sometimes in advance of the latest released version of Perl, and that certain features described here might not work on earlier versions. This section will keep track of when various features were added to Perl 5.

- In versions of Perl 5.002 prior to the gamma version, the test script in Example 1 will not function properly. You need to change the ''use lib'' line to read:

```
use lib './blib';
```

- In versions of Perl 5.002 prior to version beta 3, the line in the .xs file about ''PROTOTYPES: DISABLE'' will cause a compiler error. Simply remove that line from the file.
- In versions of Perl 5.002 prior to version 5.002b1h, the test.pl file was not automatically created by h2xs. This means that you cannot say ''make test'' to run the test script. You will need to add the following line before the ''use extension'' statement:

```
use lib './blib';
```

- In versions 5.000 and 5.001, instead of using the above line, you will need to use the following line:

```
BEGIN { unshift(@INC, "./blib") }
```

- This document assumes that the executable named ''perl'' is Perl version 5. Some systems may have installed Perl version 5 as ''perl5''.

### DYNAMIC VERSUS STATIC

It is commonly thought that if a system does not have the capability to load a library dynamically, you cannot build XSUBs. This is incorrect. You *can* build them, but you must link the XSUB's subroutines with the rest of Perl, creating a new executable. This situation is similar to Perl 4.

This tutorial can still be used on such a system. The XSUB build mechanism will check the system and build a dynamically-loadable library if possible, or else a static library and then, optionally, a new statically-linked executable with that static library linked in.

Should you wish to build a statically-linked executable on a system which can dynamically load libraries, you may, in all the following examples, where the command ''make'' with no arguments is executed, run the command ''make perl'' instead.

If you have generated such a statically-linked executable by choice, then instead of saying ''make test'', you should say ''make test_static''. On systems that cannot build dynamically-loadable libraries at all, simply saying ''make test'' is sufficient.

**EXAMPLE 1**

Our first extension will be very simple. When we call the routine in the extension, it will print out a well-known message and return.

Run `h2xs -A -n Mytest`. This creates a directory named Mytest, possibly under ext/ if that directory exists in the current working directory. Several files will be created in the Mytest dir, including MANIFEST, Makefile.PL, Mytest.pm, Mytest.xs, test.pl, and Changes.

The MANIFEST file contains the names of all the files created.

The file Makefile.PL should look something like this:

```
use ExtUtils::MakeMaker;
# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.
WriteMakefile(
    'NAME'         => 'Mytest',
    'VERSION_FROM' => 'Mytest.pm', # finds $VERSION
    'LIBS'         => [''],   # e.g., '-lm'
    'DEFINE'       => '',     # e.g., '-DHAVE_SOMETHING'
    'INC'          => '',     # e.g., '-I/usr/include/other'
);
```

The file Mytest.pm should start with something like this:

```
package Mytest;

require Exporter;
require DynaLoader;

@ISA = qw(Exporter DynaLoader);
# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.
@EXPORT = qw(

);
$VERSION = '0.01';

bootstrap Mytest $VERSION;

# Preloaded methods go here.

# Autoload methods go after __END__, and are processed by the autosplit prog

1;
__END__
# Below is the stub of documentation for your module. You better edit it!
```

And the Mytest.xs file should look something like this:

```
#ifdef __cplusplus
extern "C" {
#endif
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#ifdef __cplusplus
}
#endif

PROTOTYPES: DISABLE

MODULE = Mytest          PACKAGE = Mytest
```

Let's edit the .xs file by adding this to the end of the file:

```
void
hello()
        CODE:
        printf("Hello, world!\n");
```

Now we'll run "perl Makefile.PL". This will create a real Makefile, which make needs. Its output looks something like:

```
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Mytest
%
```

Now, running make will produce output that looks something like this (some long lines shortened for clarity):

```
% make
umask 0 && cp Mytest.pm ./blib/Mytest.pm
perl xsubpp -typemap typemap Mytest.xs >Mytest.tc && mv Mytest.tc Mytest.c
cc -c Mytest.c
Running Mkbootstrap for Mytest ()
chmod 644 Mytest.bs
LD_RUN_PATH="" ld -o ./blib/PA-RISC1.1/auto/Mytest/Mytest.sl -b Mytest.o
chmod 755 ./blib/PA-RISC1.1/auto/Mytest/Mytest.sl
cp Mytest.bs ./blib/PA-RISC1.1/auto/Mytest/Mytest.bs
chmod 644 ./blib/PA-RISC1.1/auto/Mytest/Mytest.bs
```

Now, although there is already a test.pl template ready for us, for this example only, we'll create a special test script. Create a file called hello that looks like this:

```
#! /opt/perl5/bin/perl

use ExtUtils::testlib;

use Mytest;

Mytest::hello();
```

Now we run the script and we should see the following output:

```
% perl hello
Hello, world!
%
```

**EXAMPLE 2**

Now let's add to our extension a subroutine that will take a single argument and return 1 if the argument is even, 0 if the argument is odd.

Add the following to the end of Mytest.xs:

```
int
is_even(input)
        int     input
        CODE:
        RETVAL = (input % 2 == 0);
        OUTPUT:
        RETVAL
```

There does not need to be white space at the start of the "int input" line, but it is useful for improving readability. The semi-colon at the end of that line is also optional.

Any white space may be between the "int" and "input". It is also okay for the four lines starting at the "CODE:" line to not be indented. However, for readability purposes, it is suggested that you indent them 8 spaces (or one normal tab stop).

Now rerun make to rebuild our new shared library.

Now perform the same steps as before, generating a Makefile from the Makefile.PL file, and running make.

To test that our extension works, we now need to look at the file test.pl. This file is set up to imitate the same kind of testing structure that Perl itself has. Within the test script, you perform a number of tests to confirm the behavior of the extension, printing "ok" when the test is correct, "not ok" when it is not. Change the print statement in the BEGIN block to print "1..4", and add the following code to the end of the file:

```
print &Mytest::is_even(0) == 1 ? "ok 2" : "not ok 2", "\n";
print &Mytest::is_even(1) == 0 ? "ok 3" : "not ok 3", "\n";
print &Mytest::is_even(2) == 1 ? "ok 4" : "not ok 4", "\n";
```

We will be calling the test script through the command "make test". You should see output that looks something like this:

```
% make test
PERL_DL_NONLAZY=1 /opt/perl5.002b2/bin/perl (lots of -I arguments) test.pl
1..4
ok 1
ok 2
ok 3
ok 4
%
```

**WHAT HAS GONE ON?**

The program h2xs is the starting point for creating extensions. In later examples we'll see how we can use h2xs to read header files and generate templates to connect to C routines.

h2xs creates a number of files in the extension directory. The file Makefile.PL is a perl script which will generate a true Makefile to build the extension. We'll take a closer look at it later.

The files <extension>.pm and <extension>.xs contain the meat of the extension. The .xs file holds the C routines that make up the extension. The .pm file contains routines that tell Perl how to load your extension.

Generating and invoking the Makefile created a directory blib (which stands for ‘‘build library") in the current working directory. This directory will contain the shared library that we will build. Once we have tested it, we can install it into its final location.

Invoking the test script via ‘‘make test’’ did something very important. It invoked perl with all those `-I` arguments so that it could find the various files that are part of the extension.

It is *very* important that while you are still testing extensions that you use ‘‘make test’’. If you try to run the test script all by itself, you will get a fatal error.

Another reason it is important to use ‘‘make test’’ to run your test script is that if you are testing an upgrade to an already-existing version, using ‘‘make test’’ insures that you use your new extension, not the already-existing version.

When Perl sees a `use extension;`, it searches for a file with the same name as the use’d extension that has a .pm suffix. If that file cannot be found, Perl dies with a fatal error. The default search path is contained in the `@INC` array.

In our case, Mytest.pm tells perl that it will need the Exporter and Dynamic Loader extensions. It then sets the `@ISA` and `@EXPORT` arrays and the `$VERSION` scalar; finally it tells perl to bootstrap the module. Perl will call its dynamic loader routine (if there is one) and load the shared library.

The two arrays that are set in the .pm file are very important. The `@ISA` array contains a list of other packages in which to search for methods (or subroutines) that do not exist in the current package. The `@EXPORT` array tells Perl which of the extension’s routines should be placed into the calling package’s namespace.

It’s important to select what to export carefully. Do NOT export method names and do NOT export anything else *by default* without a good reason.

As a general rule, if the module is trying to be object-oriented then don’t export anything. If it’s just a collection of functions then you can export any of the functions via another array, called `@EXPORT_OK`.

See the *perlmod* manpage for more information.

The `$VERSION` variable is used to ensure that the .pm file and the shared library are ‘‘in sync’’ with each other. Any time you make changes to the .pm or .xs files, you should increment the value of this variable.

**WRITING GOOD TEST SCRIPTS**

The importance of writing good test scripts cannot be overemphasized. You should closely follow the ‘‘ok/not ok’’ style that Perl itself uses, so that it is very easy and unambiguous to determine the outcome of each test case. When you find and fix a bug, make sure you add a test case for it.

By running ‘‘make test’’, you ensure that your test.pl script runs and uses the correct version of your extension. If you have many test cases, you might want to copy Perl’s test style. Create a directory named ‘‘t’’, and ensure all your test files end with the suffix ‘‘.t’’. The Makefile will properly run all these test files.

**EXAMPLE 3**

Our third extension will take one argument as its input, round off that value, and set the *argument* to the rounded value.

Add the following to the end of Mytest.xs:

```
void
round(arg)
        double  arg
        CODE:
        if (arg > 0.0) {
                arg = floor(arg + 0.5);
        } else if (arg < 0.0) {
                arg = ceil(arg - 0.5);
        } else {
                arg = 0.0;
        }
        OUTPUT:
        arg
```

Edit the Makefile.PL file so that the corresponding line looks like this:

```
'LIBS'      => ['-lm'],   # e.g., '-lm'
```

Generate the Makefile and run make. Change the BEGIN block to print out "1..9" and add the following to test.pl:

```
$i = -1.5; &Mytest::round($i); print $i == -2.0 ? "ok 5" : "not ok 5", "\n";
$i = -1.1; &Mytest::round($i); print $i == -1.0 ? "ok 6" : "not ok 6", "\n";
$i = 0.0; &Mytest::round($i); print $i == 0.0 ? "ok 7" : "not ok 7", "\n";
$i = 0.5; &Mytest::round($i); print $i == 1.0 ? "ok 8" : "not ok 8", "\n";
$i = 1.2; &Mytest::round($i); print $i == 1.0 ? "ok 9" : "not ok 9", "\n";
```

Running "make test" should now print out that all nine tests are okay.

You might be wondering if you can round a constant. To see what happens, add the following line to test.pl temporarily:

```
&Mytest::round(3);
```

Run "make test" and notice that Perl dies with a fatal error. Perl won't let you change the value of constants!

**WHAT'S NEW HERE?**

Two things are new here. First, we've made some changes to Makefile.PL. In this case, we've specified an extra library to link in, the math library libm. We'll talk later about how to write XSUBs that can call every routine in a library.

Second, the value of the function is being passed back not as the function's return value, but through the same variable that was passed into the function.

**INPUT AND OUTPUT PARAMETERS**

You specify the parameters that will be passed into the XSUB just after you declare the function return value and name. Each parameter line starts with optional white space, and may have an optional terminating semicolon.

The list of output parameters occurs after the OUTPUT: directive. The use of RETVAL tells Perl that you wish to send this value back as the return value of the XSUB function. In Example 3, the value we wanted returned was contained in the same variable we passed in, so we listed it (and not RETVAL) in the OUTPUT: section.

**THE XSUBPP COMPILER**

The compiler xsubpp takes the XS code in the .xs file and converts it into C code, placing it in a file whose suffix is .c. The C code created makes heavy use of the C functions within Perl.

**THE TYPEMAP FILE**

The xsubpp compiler uses rules to convert from Perl's data types (scalar, array, etc.) to C's data types (int, char *, etc.). These rules are stored in the typemap file ($PERLLIB/ExtUtils/typemap). This file is split into three parts.

The first part attempts to map various C data types to a coded flag, which has some correspondence with the various Perl types. The second part contains C code which xsubpp uses for input parameters. The third part contains C code which xsubpp uses for output parameters. We'll talk more about the C code later.

Let's now take a look at a portion of the .c file created for our extension.

```
XS(XS_Mytest_round)
{
    dXSARGS;
    if (items != 1)
        croak("Usage: Mytest::round(arg)");
    {
        double  arg = (double)SvNV(ST(0));      /* XXXXX */
        if (arg > 0.0) {
                arg = floor(arg + 0.5);
        } else if (arg < 0.0) {
                arg = ceil(arg - 0.5);
        } else {
                arg = 0.0;
        }
        sv_setnv(ST(0), (double)arg);           /* XXXXX */
    }
    XSRETURN(1);
}
```

Notice the two lines marked with "XXXXX". If you check the first section of the typemap file, you'll see that doubles are of type T_DOUBLE. In the INPUT section, an argument that is T_DOUBLE is assigned to the variable arg by calling the routine SvNV on something, then casting it to double, then assigned to the variable arg. Similarly, in the OUTPUT section, once arg has its final value, it is passed to the sv_setnv function to be passed back to the calling subroutine. These two functions are explained in the *perlguts* manpage; we'll talk more later about what that "*ST*(0)" means in the section on the argument stack.

**WARNING**

In general, it's not a good idea to write extensions that modify their input parameters, as in Example 3. However, to accommodate better calling pre-existing C routines, which often do modify their input parameters, this behavior is tolerated. The next example will show how to do this.

**EXAMPLE 4**

In this example, we'll now begin to write XSUBs that will interact with predefined C libraries. To begin with, we will build a small library of our own, then let h2xs write our .pm and .xs files for us.

Create a new directory called Mytest2 at the same level as the directory Mytest. In the Mytest2 directory, create another directory called mylib, and cd into that directory.

Here we'll create some files that will generate a test library. These will include a C source file and a header file. We'll also create a Makefile.PL in this directory. Then we'll make sure that running make at the Mytest2 level will automatically run this Makefile.PL file and the resulting Makefile.

In the testlib directory, create a file mylib.h that looks like this:

```
#define TESTVAL 4

extern double   foo(int, long, const char*);
```

Also create a file mylib.c that looks like this:

```
#include <stdlib.h>
#include "./mylib.h"

double
foo(a, b, c)
int             a;
long            b;
const char *    c;
{
        return (a + b + atof(c) + TESTVAL);
}
```

And finally create a file Makefile.PL that looks like this:

```
use ExtUtils::MakeMaker;
$Verbose = 1;
WriteMakefile(
    NAME      => 'Mytest2::mylib',
    SKIP      => [qw(all static static_lib dynamic dynamic_lib)],
    clean     => {'FILES' => 'libmylib$(LIB_EXT)'},
);

sub MY::top_targets {
        '
all :: static

static ::       libmylib$(LIB_EXT)

libmylib$(LIB_EXT): $(O_FILES)
        $(AR) cr libmylib$(LIB_EXT) $(O_FILES)
        $(RANLIB) libmylib$(LIB_EXT)

';
}
```

We will now create the main top-level Mytest2 files. Change to the directory above Mytest2 and run the following command:

```
% h2xs -O -n Mytest2 ./Mytest2/mylib/mylib.h
```

This will print out a warning about overwriting Mytest2, but that's okay. Our files are stored in Mytest2/mylib, and will be untouched.

The normal Makefile.PL that h2xs generates doesn't know about the mylib directory. We need to tell it that there is a subdirectory and that we will be generating a library in it. Let's add the following key-value pair to the WriteMakefile call:

```
'MYEXTLIB' => 'mylib/libmylib$(LIB_EXT)',
```

and a new replacement subroutine too:

```
sub MY::postamble {
'
$(MYEXTLIB): mylib/Makefile
        cd mylib && $(MAKE) $(PASTHRU)
';
}
```

(Note: Most makes will require that there be a tab character that indents the line `cd mylib && $(MAKE) $(PASTHRU)`, similarly for the Makefile in the subdirectory.)

Let's also fix the MANIFEST file so that it accurately reflects the contents of our extension. The single line that says ‘‘mylib’’ should be replaced by the following three lines:

```
mylib/Makefile.PL
mylib/mylib.c
mylib/mylib.h
```

To keep our namespace nice and unpolluted, edit the .pm file and change the lines setting `@EXPORT` to `@EXPORT_OK` (there are two: one in the line beginning ‘‘use vars’’ and one setting the array itself). Finally, in the .xs file, edit the #include line to read:

```
#include "mylib/mylib.h"
```

And also add the following function definition to the end of the .xs file:

```
double
foo(a,b,c)
        int             a
        long            b
        const char *    c
        OUTPUT:
        RETVAL
```

Now we also need to create a typemap file because the default Perl doesn't currently support the const char * type. Create a file called typemap and place the following in it:

```
const char *    T_PV
```

Now run perl on the top-level Makefile.PL. Notice that it also created a Makefile in the mylib directory. Run make and see that it does cd into the mylib directory and run make in there as well.

Now edit the test.pl script and change the BEGIN block to print ‘‘1..4’’, and add the following lines to the end of the script:

```
print &Mytest2::foo(1, 2, "Hello, world!") == 7 ? "ok 2\n" : "not ok 2\n";
print &Mytest2::foo(1, 2, "0.0") == 7 ? "ok 3\n" : "not ok 3\n";
print abs(&Mytest2::foo(0, 0, "-3.4") - 0.6) <= 0.01 ? "ok 4\n" : "not ok 4\
```

(When dealing with floating-point comparisons, it is often useful not to check for equality, but rather the difference being below a certain epsilon factor, 0.01 in this case)

Run ‘‘make test’’ and all should be well.

**WHAT HAS HAPPENED HERE?**

Unlike previous examples, we've now run h2xs on a real include file. This has caused some extra goodies to appear in both the .pm and .xs files.

- In the .xs file, there's now a #include declaration with the full path to the mylib.h header file.
- There's now some new C code that's been added to the .xs file. The purpose of the `constant` routine is to make the values that are #define'd in the header file available to the Perl script (in this case, by calling `&main::TESTVAL`). There's also some XS code to allow calls to the `constant` routine.
- The .pm file has exported the name TESTVAL in the `@EXPORT` array. This could lead to name clashes. A good rule of thumb is that if the #define is going to be used by only the C routines themselves, and not by the user, they should be removed from the `@EXPORT` array. Alternately, if you don't mind using the "fully qualified name" of a variable, you could remove most or all of the items in the `@EXPORT` array.
- If our include file contained #include directives, these would not be processed at all by h2xs. There is no good solution to this right now.

We've also told Perl about the library that we built in the mylib subdirectory. That required the addition of only the MYEXTLIB variable to the WriteMakefile call and the replacement of the postamble subroutine to cd into the subdirectory and run make. The Makefile.PL for the library is a bit more complicated, but not excessively so. Again we replaced the postamble subroutine to insert our own code. This code specified simply that the library to be created here was a static archive (as opposed to a dynamically loadable library) and provided the commands to build it.

**SPECIFYING ARGUMENTS TO XSUBPP**

With the completion of Example 4, we now have an easy way to simulate some real-life libraries whose interfaces may not be the cleanest in the world. We shall now continue with a discussion of the arguments passed to the xsubpp compiler.

When you specify arguments in the .xs file, you are really passing three pieces of information for each one listed. The first piece is the order of that argument relative to the others (first, second, etc). The second is the type of argument, and consists of the type declaration of the argument (e.g., int, char*, etc). The third piece is the exact way in which the argument should be used in the call to the library function from this XSUB. This would mean whether or not to place a "&" before the argument or not, meaning the argument expects to be passed the address of the specified data type.

There is a difference between the two arguments in this hypothetical function:

```
int
foo(a,b)
        char    &a
        char *  b
```

The first argument to this function would be treated as a char and assigned to the variable a, and its address would be passed into the function foo. The second argument would be treated as a string pointer and assigned to the variable b. The *value* of b would be passed into the function foo. The actual call to the function foo that xsubpp generates would look like this:

```
foo(&a, b);
```

Xsubpp will identically parse the following function argument lists:

```
char    &a
char&a
char    & a
```

However, to help ease understanding, it is suggested that you place a "&" next to the variable name and away from the variable type), and place a "*" near the variable type, but away from the variable name (as in the complete example above). By doing so, it is easy to understand exactly what will be passed to the C function -- it will be whatever is in the "last column".

You should take great pains to try to pass the function the type of variable it wants, when possible. It will save you a lot of trouble in the long run.

#### THE ARGUMENT STACK

If we look at any of the C code generated by any of the examples except example 1, you will notice a number of references to *ST*(n), where n is usually 0. The "ST" is actually a macro that points to the n'th argument on the argument stack. *ST*(0) is thus the first argument passed to the XSUB, *ST*(1) is the second argument, and so on.

When you list the arguments to the XSUB in the .xs file, that tells xsubpp which argument corresponds to which of the argument stack (i.e., the first one listed is the first argument, and so on). You invite disaster if you do not list them in the same order as the function expects them.

#### EXTENDING YOUR EXTENSION

Sometimes you might want to provide some extra methods or subroutines to assist in making the interface between Perl and your extension simpler or easier to understand. These routines should live in the .pm file. Whether they are automatically loaded when the extension itself is loaded or loaded only when called depends on where in the .pm file the subroutine definition is placed.

#### DOCUMENTING YOUR EXTENSION

There is absolutely no excuse for not documenting your extension. Documentation belongs in the .pm file. This file will be fed to pod2man, and the embedded documentation will be converted to the manpage format, then placed in the blib directory. It will be copied to Perl's man page directory when the extension is installed.

You may intersperse documentation and Perl code within the .pm file. In fact, if you want to use method autoloading, you must do this, as the comment inside the .pm file explains.

See the *perlpod* manpage for more information about the pod format.

#### INSTALLING YOUR EXTENSION

Once your extension is complete and passes all its tests, installing it is quite simple: you simply run "make install". You will either need to have write permission into the directories where Perl is installed, or ask your system administrator to run the make for you.

#### SEE ALSO

For more information, consult the *perlguts* manpage, the *perlxs* manpage, the *perlmod* manpage, and the *perlpod* manpage.

### Author

Jeff Okamoto <*okamoto@corp.hp.com*>

Reviewed and assisted by Dean Roehrich, Ilya Zakharevich, Andreas Koenig, and Tim Bunce.

### Last Changed

1996/7/10

## NAME

perlguts − Perl's Internal Functions

## DESCRIPTION

This document attempts to describe some of the internal functions of the Perl executable. It is far from complete and probably contains many errors. Please refer any questions or comments to the author below.

## Variables

### Datatypes

Perl has three typedefs that handle Perl's three main data types:

```
SV  Scalar Value
AV  Array Value
HV  Hash Value
```

Each typedef has specific routines that manipulate the various data types.

### What is an "IV"?

Perl uses a special typedef IV which is a simple integer type that is guaranteed to be large enough to hold a pointer (as well as an integer).

Perl also uses two special typedefs, I32 and I16, which will always be at least 32-bits and 16-bits long, respectively.

### Working with SVs

An SV can be created and loaded with one command. There are four types of values that can be loaded: an integer value (IV), a double (NV), a string, (PV), and another scalar (SV).

The six routines are:

```
SV*  newSViv(IV);
SV*  newSVnv(double);
SV*  newSVpv(char*, int);
SV*  newSVpvn(char*, int);
SV*  newSVpvf(const char*, ...);
SV*  newSVsv(SV*);
```

To change the value of an *already-existing* SV, there are seven routines:

```
void  sv_setiv(SV*, IV);
void  sv_setuv(SV*, UV);
void  sv_setnv(SV*, double);
void  sv_setpv(SV*, char*);
void  sv_setpvn(SV*, char*, int)
void  sv_setpvf(SV*, const char*, ...);
void  sv_setpvfn(SV*, const char*, STRLEN, va_list *, SV **, I32, bool);
void  sv_setsv(SV*, SV*);
```

Notice that you can choose to specify the length of the string to be assigned by using `sv_setpvn`, `newSVpvn`, or `newSVpv`, or you may allow Perl to calculate the length by using `sv_setpv` or by specifying 0 as the second argument to `newSVpv`. Be warned, though, that Perl will determine the string's length by using `strlen`, which depends on the string terminating with a NUL character.

The arguments of `sv_setpvf` are processed like `sprintf`, and the formatted output becomes the value.

`sv_setpvfn` is an analogue of `vsprintf`, but it allows you to specify either a pointer to a variable argument list or the address and length of an array of SVs. The last argument points to a boolean; on return, if that boolean is true, then locale-specific information has been used to format the string, and the string's contents are therefore untrustworty (see the *perlsec* manpage). This pointer may be NULL if that information is not important. Note that this function requires you to specify the length of the format.

The `sv_set*()` functions are not generic enough to operate on values that have "magic". See the section on *Magic Virtual Tables* later in this document.

All SVs that contain strings should be terminated with a NUL character. If it is not NUL−terminated there is a risk of core dumps and corruptions from code which passes the string to C functions or system calls which expect a NUL−terminated string. Perl's own functions typically add a trailing NUL for this reason. Nevertheless, you should be very careful when you pass a string stored in an SV to a C function or system call.

To access the actual value that an SV points to, you can use the macros:

```
SvIV(SV*)
SvNV(SV*)
SvPV(SV*, STRLEN len)
```

which will automatically coerce the actual scalar type into an IV, double, or string.

In the `SvPV` macro, the length of the string returned is placed into the variable `len` (this is a macro, so you do *not* use `&len`). If you do not care what the length of the data is, use the global variable `PL_na`. Remember, however, that Perl allows arbitrary strings of data that may both contain NULs and might not be terminated by a NUL.

If you want to know if the scalar value is TRUE, you can use:

```
SvTRUE(SV*)
```

Although Perl will automatically grow strings for you, if you need to force Perl to allocate more memory for your SV, you can use the macro

```
SvGROW(SV*, STRLEN newlen)
```

which will determine if more memory needs to be allocated. If so, it will call the function `sv_grow`. Note that `SvGROW` can only increase, not decrease, the allocated memory of an SV and that it does not automatically add a byte for the a trailing NUL (perl's own string functions typically do `SvGROW(sv, len + 1)`).

If you have an SV and want to know what kind of data Perl thinks is stored in it, you can use the following macros to check the type of SV you have.

```
SvIOK(SV*)
SvNOK(SV*)
SvPOK(SV*)
```

You can get and set the current length of the string stored in an SV with the following macros:

```
SvCUR(SV*)
SvCUR_set(SV*, I32 val)
```

You can also get a pointer to the end of the string stored in the SV with the macro:

```
SvEND(SV*)
```

But note that these last three macros are valid only if `SvPOK()` is true.

If you want to append something to the end of string stored in an `SV*`, you can use the following functions:

```
void  sv_catpv(SV*, char*);
void  sv_catpvn(SV*, char*, int);
void  sv_catpvf(SV*, const char*, ...);
void  sv_catpvfn(SV*, const char*, STRLEN, va_list *, SV **, I32, bool);
void  sv_catsv(SV*, SV*);
```

The first function calculates the length of the string to be appended by using `strlen`. In the second, you specify the length of the string yourself. The third function processes its arguments like `sprintf` and appends the formatted output. The fourth function works like `vsprintf`. You can specify the address and length of an array of SVs instead of the va_list argument. The fifth function extends the string stored in the first SV with the string stored in the second SV. It also forces the second SV to be interpreted as a string.

The `sv_cat*()` functions are not generic enough to operate on values that have "magic". See the section on *Magic Virtual Tables* later in this document.

If you know the name of a scalar variable, you can get a pointer to its SV by using the following:

```
SV*  perl_get_sv("package::varname", FALSE);
```

This returns NULL if the variable does not exist.

If you want to know if this variable (or any other SV) is actually `defined`, you can call:

```
SvOK(SV*)
```

The scalar `undef` value is stored in an SV instance called `PL_sv_undef`. Its address can be used whenever an `SV*` is needed.

There are also the two values `PL_sv_yes` and `PL_sv_no`, which contain Boolean TRUE and FALSE values, respectively. Like `PL_sv_undef`, their addresses can be used whenever an `SV*` is needed.

Do not be fooled into thinking that `(SV *) 0` is the same as `&PL_sv_undef`. Take this code:

```
SV* sv = (SV*) 0;
if (I-am-to-return-a-real-value) {
        sv = sv_2mortal(newSViv(42));
}
sv_setsv(ST(0), sv);
```

This code tries to return a new SV (which contains the value 42) if it should return a real value, or undef otherwise. Instead it has returned a NULL pointer which, somewhere down the line, will cause a segmentation violation, bus error, or just weird results. Change the zero to `&PL_sv_undef` in the first line and all will be well.

To free an SV that you've created, call `SvREFCNT_dec(SV*)`. Normally this call is not necessary (see the section on *Reference Counts and Mortality*).

### What's Really Stored in an SV?

Recall that the usual method of determining the type of scalar you have is to use `Sv*OK` macros. Because a scalar can be both a number and a string, usually these macros will always return TRUE and calling the `Sv*V` macros will do the appropriate conversion of string to integer/double or integer/double to string.

If you *really* need to know if you have an integer, double, or string pointer in an SV, you can use the following three macros instead:

```
SvIOKp(SV*)
SvNOKp(SV*)
SvPOKp(SV*)
```

These will tell you if you truly have an integer, double, or string pointer stored in your SV. The "p" stands for private.

In general, though, it's best to use the `Sv*V` macros.

### Working with AVs

There are two ways to create and load an AV. The first method creates an empty AV:

```
AV*  newAV();
```

The second method both creates the AV and initially populates it with SVs:

```
AV*  av_make(I32 num, SV **ptr);
```

The second argument points to an array containing `num` `SV*`'s. Once the AV has been created, the SVs can be destroyed, if so desired.

Once the AV has been created, the following operations are possible on AVs:

```
void  av_push(AV*, SV*);
SV*   av_pop(AV*);
SV*   av_shift(AV*);
void  av_unshift(AV*, I32 num);
```

These should be familiar operations, with the exception of `av_unshift`. This routine adds `num` elements at the front of the array with the `undef` value. You must then use `av_store` (described below) to assign values to these new elements.

Here are some other functions:

```
I32   av_len(AV*);
SV**  av_fetch(AV*, I32 key, I32 lval);
SV**  av_store(AV*, I32 key, SV* val);
```

The `av_len` function returns the highest index value in array (just like $#array in Perl). If the array is empty, −1 is returned. The `av_fetch` function returns the value at index `key`, but if `lval` is non-zero, then `av_fetch` will store an undef value at that index. The `av_store` function stores the value `val` at index `key`, and does not increment the reference count of `val`. Thus the caller is responsible for taking care of that, and if `av_store` returns NULL, the caller will have to decrement the reference count to avoid a memory leak. Note that `av_fetch` and `av_store` both return `SV**`'s, not `SV*`'s as their return value.

```
void  av_clear(AV*);
void  av_undef(AV*);
void  av_extend(AV*, I32 key);
```

The `av_clear` function deletes all the elements in the AV* array, but does not actually delete the array itself. The `av_undef` function will delete all the elements in the array plus the array itself. The `av_extend` function extends the array so that it contains `key` elements. If `key` is less than the current length of the array, then nothing is done.

If you know the name of an array variable, you can get a pointer to its AV by using the following:

```
AV*  perl_get_av("package::varname", FALSE);
```

This returns NULL if the variable does not exist.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use the array access functions on tied arrays.

**Working with HVs**

To create an HV, you use the following routine:

```
HV*  newHV();
```

Once the HV has been created, the following operations are possible on HVs:

```
SV**  hv_store(HV*, char* key, U32 klen, SV* val, U32 hash);
SV**  hv_fetch(HV*, char* key, U32 klen, I32 lval);
```

The `klen` parameter is the length of the key being passed in (Note that you cannot pass 0 in as a value of `klen` to tell Perl to measure the length of the key). The `val` argument contains the SV pointer to the scalar being stored, and `hash` is the precomputed hash value (zero if you want `hv_store` to calculate it for you). The `lval` parameter indicates whether this fetch is actually a part of a store operation, in which case a new undefined value will be added to the HV with the supplied key and `hv_fetch` will return as if the value had already existed.

Remember that `hv_store` and `hv_fetch` return `SV**`'s and not just `SV*`. To access the scalar value, you must first dereference the return value. However, you should check to make sure that the return value is not NULL before dereferencing it.

These two functions check if a hash table entry exists, and deletes it.

```
bool  hv_exists(HV*, char* key, U32 klen);
SV*   hv_delete(HV*, char* key, U32 klen, I32 flags);
```

If `flags` does not include the `G_DISCARD` flag then `hv_delete` will create and return a mortal copy of the deleted value.

And more miscellaneous functions:

```
void   hv_clear(HV*);
void   hv_undef(HV*);
```

Like their AV counterparts, `hv_clear` deletes all the entries in the hash table but does not actually delete the hash table. The `hv_undef` deletes both the entries and the hash table itself.

Perl keeps the actual data in linked list of structures with a typedef of HE. These contain the actual key and value pointers (plus extra administrative overhead). The key is a string pointer; the value is an `SV*`. However, once you have an `HE*`, to get the actual key and value, use the routines specified below.

```
I32    hv_iterinit(HV*);
        /* Prepares starting point to traverse hash table */
HE*    hv_iternext(HV*);
        /* Get the next entry, and return a pointer to a
           structure that has both the key and value */
char*  hv_iterkey(HE* entry, I32* retlen);
        /* Get the key from an HE structure and also return
           the length of the key string */
SV*    hv_iterval(HV*, HE* entry);
        /* Return a SV pointer to the value of the HE
           structure */
SV*    hv_iternextsv(HV*, char** key, I32* retlen);
        /* This convenience routine combines hv_iternext,
           hv_iterkey, and hv_iterval.  The key and retlen
           arguments are return values for the key and its
           length.  The value is returned in the SV* argument */
```

If you know the name of a hash variable, you can get a pointer to its HV by using the following:

```
HV*  perl_get_hv("package::varname", FALSE);
```

This returns NULL if the variable does not exist.

The hash algorithm is defined in the `PERL_HASH(hash, key, klen)` macro:

```
i = klen;
hash = 0;
s = key;
while (i--)
    hash = hash * 33 + *s++;
```

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use the hash access functions on tied hashes.

### Hash API Extensions

Beginning with version 5.004, the following functions are also supported:

```
HE*     hv_fetch_ent  (HV* tb, SV* key, I32 lval, U32 hash);
HE*     hv_store_ent  (HV* tb, SV* key, SV* val, U32 hash);

bool    hv_exists_ent (HV* tb, SV* key, U32 hash);
SV*     hv_delete_ent (HV* tb, SV* key, I32 flags, U32 hash);

SV*     hv_iterkeysv  (HE* entry);
```

Note that these functions take `SV*` keys, which simplifies writing of extension code that deals with hash structures. These functions also allow passing of `SV*` keys to `tie` functions without forcing you to stringify the keys (unlike the previous set of functions).

They also return and accept whole hash entries (`HE*`), making their use more efficient (since the hash number for a particular string doesn't have to be recomputed every time). See the section on *API LISTING* later in this document for detailed descriptions.

The following macros must always be used to access the contents of hash entries. Note that the arguments to these macros must be simple variables, since they may get evaluated more than once. See the section on *API LISTING* later in this document for detailed descriptions of these macros.

```
HePV(HE* he, STRLEN len)
HeVAL(HE* he)
HeHASH(HE* he)
HeSVKEY(HE* he)
HeSVKEY_force(HE* he)
HeSVKEY_set(HE* he, SV* sv)
```

These two lower level macros are defined, but must only be used when dealing with keys that are not `SV*`s:

```
HeKEY(HE* he)
HeKLEN(HE* he)
```

Note that both `hv_store` and `hv_store_ent` do not increment the reference count of the stored `val`, which is the caller's responsibility. If these functions return a NULL value, the caller will usually have to decrement the reference count of `val` to avoid a memory leak.

**References**

References are a special type of scalar that point to other data types (including references).

To create a reference, use either of the following functions:

```
SV* newRV_inc((SV*) thing);
SV* newRV_noinc((SV*) thing);
```

The `thing` argument can be any of an `SV*`, `AV*`, or `HV*`. The functions are identical except that `newRV_inc` increments the reference count of the `thing`, while `newRV_noinc` does not. For historical reasons, `newRV` is a synonym for `newRV_inc`.

Once you have a reference, you can use the following macro to dereference the reference:

```
SvRV(SV*)
```

then call the appropriate routines, casting the returned `SV*` to either an `AV*` or `HV*`, if required.

To determine if an SV is a reference, you can use the following macro:

```
SvROK(SV*)
```

To discover what type of value the reference refers to, use the following macro and then check the return value.

```
SvTYPE(SvRV(SV*))
```

The most useful types that will be returned are:

```
SVt_IV    Scalar
SVt_NV    Scalar
SVt_PV    Scalar
SVt_RV    Scalar
SVt_PVAV  Array
SVt_PVHV  Hash
SVt_PVCV  Code
SVt_PVGV  Glob (possible a file handle)
SVt_PVMG  Blessed or Magical Scalar

See the sv.h header file for more details.
```

**Blessed References and Class Objects**

References are also used to support object-oriented programming. In the OO lexicon, an object is simply a reference that has been blessed into a package (or class). Once blessed, the programmer may now use the reference to access the various methods in the class.

A reference can be blessed into a package with the following function:

```
SV* sv_bless(SV* sv, HV* stash);
```

The `sv` argument must be a reference. The `stash` argument specifies which class the reference will belong to. See the section on *Stashes and Globs* for information on converting class names into stashes.

/* Still under construction */

Upgrades rv to reference if not already one. Creates new SV for rv to point to. If `classname` is non-null, the SV is blessed into the specified class. SV is returned.

```
    SV* newSVrv(SV* rv, char* classname);
```

Copies integer or double into an SV whose reference is `rv`. SV is blessed if `classname` is non-null.

```
SV* sv_setref_iv(SV* rv, char* classname, IV iv);
SV* sv_setref_nv(SV* rv, char* classname, NV iv);
```

Copies the pointer value (*the address, not the string!*) into an SV whose reference is rv. SV is blessed if `classname` is non-null.

```
SV* sv_setref_pv(SV* rv, char* classname, PV iv);
```

Copies string into an SV whose reference is `rv`. Set length to 0 to let Perl calculate the string length. SV is blessed if `classname` is non-null.

```
SV* sv_setref_pvn(SV* rv, char* classname, PV iv, int length);
```

Tests whether the SV is blessed into the specified class. It does not check inheritance relationships.

```
int  sv_isa(SV* sv, char* name);
```

Tests whether the SV is a reference to a blessed object.

```
int  sv_isobject(SV* sv);
```

Tests whether the SV is derived from the specified class. SV can be either a reference to a blessed object or a string containing a class name. This is the function implementing the `UNIVERSAL::isa` functionality.

```
bool sv_derived_from(SV* sv, char* name);
```

To check if you've got an object derived from a specific class you have to write:

```
if (sv_isobject(sv) && sv_derived_from(sv, class)) { ... }
```

### Creating New Variables

To create a new Perl variable with an undef value which can be accessed from your Perl script, use the following routines, depending on the variable type.

```
SV*  perl_get_sv("package::varname", TRUE);
AV*  perl_get_av("package::varname", TRUE);
HV*  perl_get_hv("package::varname", TRUE);
```

Notice the use of TRUE as the second parameter. The new variable can now be set, using the routines appropriate to the data type.

There are additional macros whose values may be bitwise OR'ed with the `TRUE` argument to enable certain extra features. Those bits are:

```
GV_ADDMULTI Marks the variable as multiply defined, thus preventing the
            "Name <varname> used only once: possible typo" warning.
GV_ADDWARN  Issues the warning "Had to create <varname> unexpectedly" if
            the variable did not exist before the function was called.
```

If you do not specify a package name, the variable is created in the current package.

### Reference Counts and Mortality

Perl uses an reference count-driven garbage collection mechanism. SVs, AVs, or HVs (xV for short in the following) start their life with a reference count of 1. If the reference count of an xV ever drops to 0, then it will be destroyed and its memory made available for reuse.

This normally doesn't happen at the Perl level unless a variable is undef'ed or the last variable holding a reference to it is changed or overwritten. At the internal level, however, reference counts can be manipulated with the following macros:

```
int SvREFCNT(SV* sv);
SV* SvREFCNT_inc(SV* sv);
void SvREFCNT_dec(SV* sv);
```

However, there is one other function which manipulates the reference count of its argument. The `newRV_inc` function, you will recall, creates a reference to the specified argument. As a side effect, it increments the argument's reference count. If this is not what you want, use `newRV_noinc` instead.

For example, imagine you want to return a reference from an XSUB function. Inside the XSUB routine, you create an SV which initially has a reference count of one. Then you call `newRV_inc`, passing it the just-created SV. This returns the reference as a new SV, but the reference count of the SV you passed to `newRV_inc` has been incremented to two. Now you return the reference from the XSUB routine and forget about the SV. But Perl hasn't! Whenever the returned reference is destroyed, the reference count of the original SV is decreased to one and nothing happens. The SV will hang around without any way to access it until Perl itself terminates. This is a memory leak.

The correct procedure, then, is to use `newRV_noinc` instead of `newRV_inc`. Then, if and when the last reference is destroyed, the reference count of the SV will go to zero and it will be destroyed, stopping any memory leak.

There are some convenience functions available that can help with the destruction of xVs. These functions introduce the concept of "mortality". An xV that is mortal has had its reference count marked to be decremented, but not actually decremented, until "a short time later". Generally the term "short time later" means a single Perl statement, such as a call to an XSUB function. The actual determinant for when mortal xVs have their reference count decremented depends on two macros, SAVETMPS and FREETMPS. See the *perlcall* manpage and the *perlxs* manpage for more details on these macros.

"Mortalization" then is at its simplest a deferred `SvREFCNT_dec`. However, if you mortalize a variable twice, the reference count will later be decremented twice.

You should be careful about creating mortal variables. Strange things can happen if you make the same value mortal within multiple contexts, or if you make a variable mortal multiple times.

To create a mortal variable, use the functions:

```
SV*  sv_newmortal()
SV*  sv_2mortal(SV*)
SV*  sv_mortalcopy(SV*)
```

The first call creates a mortal SV, the second converts an existing SV to a mortal SV (and thus defers a call to `SvREFCNT_dec`), and the third creates a mortal copy of an existing SV.

The mortal routines are not just for SVs -- AVs and HVs can be made mortal by passing their address (type-casted to `SV*`) to the `sv_2mortal` or `sv_mortalcopy` routines.

### Stashes and Globs

A "stash" is a hash that contains all of the different objects that are contained within a package. Each key of the stash is a symbol name (shared by all the different types of objects that have the same name), and each value in the hash table is a GV (Glob Value). This GV in turn contains references to the various objects of that name, including (but not limited to) the following:

```
Scalar Value
Array Value
Hash Value
I/O Handle
Format
Subroutine
```

There is a single stash called "PL_defstash" that holds the items that exist in the "main" package. To get at the items in other packages, append the string "::" to the package name. The items in the "Foo" package are in the stash "Foo::" in PL_defstash. The items in the "Bar::Baz" package are in the stash "Baz::" in "Bar::"'s stash.

To get the stash pointer for a particular package, use the function:

```
HV*  gv_stashpv(char* name, I32 create)
HV*  gv_stashsv(SV*, I32 create)
```

The first function takes a literal string, the second uses the string stored in the SV. Remember that a stash is just a hash table, so you get back an `HV*`. The `create` flag will create a new package if it is set.

The name that `gv_stash*v` wants is the name of the package whose symbol table you want. The default package is called `main`. If you have multiply nested packages, pass their names to `gv_stash*v`, separated by `::` as in the Perl language itself.

Alternately, if you have an SV that is a blessed reference, you can find out the stash pointer by using:

```
HV*  SvSTASH(SvRV(SV*));
```

then use the following to get the package name itself:

```
char*  HvNAME(HV* stash);
```

If you need to bless or re-bless an object you can use the following function:

```
SV*  sv_bless(SV*, HV* stash)
```

where the first argument, an `SV*`, must be a reference, and the second argument is a stash. The returned `SV*` can now be used in the same way as any other SV.

For more information on references and blessings, consult the *perlref* manpage.

**Double-Typed SVs**

Scalar variables normally contain only one type of value, an integer, double, pointer, or reference. Perl will automatically convert the actual scalar data from the stored type into the requested type.

Some scalar variables contain more than one type of scalar data. For example, the variable `$!` contains either the numeric value of `errno` or its string equivalent from either `strerror` or `sys_errlist[]`.

To force multiple data values into an SV, you must do two things: use the `sv_set*v` routines to add the additional scalar type, then set a flag so that Perl will believe it contains more than one type of data. The four macros to set the flags are:

```
    SvIOK_on
    SvNOK_on
    SvPOK_on
    SvROK_on
```

The particular macro you must use depends on which `sv_set*v` routine you called first. This is because every `sv_set*v` routine turns on only the bit for the particular type of data being set, and turns off all the rest.

For example, to create a new Perl variable called "dberror" that contains both the numeric and descriptive string error values, you could use the following code:

```
extern int  dberror;
extern char *dberror_list;
```

```
SV* sv = perl_get_sv("dberror", TRUE);
sv_setiv(sv, (IV) dberror);
sv_setpv(sv, dberror_list[dberror]);
SvIOK_on(sv);
```

If the order of `sv_setiv` and `sv_setpv` had been reversed, then the macro `SvPOK_on` would need to be called instead of `SvIOK_on`.

### Magic Variables

[This section still under construction. Ignore everything here. Post no bills. Everything not permitted is forbidden.]

Any SV may be magical, that is, it has special features that a normal SV does not have. These features are stored in the SV structure in a linked list of `struct magic`'s, typedef'ed to `MAGIC`.

```
struct magic {
    MAGIC*      mg_moremagic;
    MGVTBL*     mg_virtual;
    U16         mg_private;
    char        mg_type;
    U8          mg_flags;
    SV*         mg_obj;
    char*       mg_ptr;
    I32         mg_len;
};
```

Note this is current as of patchlevel 0, and could change at any time.

### Assigning Magic

Perl adds magic to an SV using the sv_magic function:

```
void sv_magic(SV* sv, SV* obj, int how, char* name, I32 namlen);
```

The `sv` argument is a pointer to the SV that is to acquire a new magical feature.

If `sv` is not already magical, Perl uses the `SvUPGRADE` macro to set the `SVt_PVMG` flag for the `sv`. Perl then continues by adding it to the beginning of the linked list of magical features. Any prior entry of the same type of magic is deleted. Note that this can be overridden, and multiple instances of the same type of magic can be associated with an SV.

The `name` and `namlen` arguments are used to associate a string with the magic, typically the name of a variable. `namlen` is stored in the `mg_len` field and if `name` is non-null and `namlen` >= 0 a malloc'd copy of the name is stored in `mg_ptr` field.

The sv_magic function uses `how` to determine which, if any, predefined "Magic Virtual Table" should be assigned to the `mg_virtual` field. See the "Magic Virtual Table" section below. The `how` argument is also stored in the `mg_type` field.

The `obj` argument is stored in the `mg_obj` field of the `MAGIC` structure. If it is not the same as the `sv` argument, the reference count of the `obj` object is incremented. If it is the same, or if the `how` argument is "#", or if it is a NULL pointer, then `obj` is merely stored, without the reference count being incremented.

There is also a function to add magic to an `HV`:

```
void hv_magic(HV *hv, GV *gv, int how);
```

This simply calls `sv_magic` and coerces the `gv` argument into an `SV`.

To remove the magic from an SV, call the function sv_unmagic:

```
void sv_unmagic(SV *sv, int type);
```

The `type` argument should be equal to the `how` value when the `SV` was initially made magical.

### Magic Virtual Tables

The `mg_virtual` field in the `MAGIC` structure is a pointer to a `MGVTBL`, which is a structure of function pointers and stands for "Magic Virtual Table" to handle the various operations that might be applied to that variable.

The `MGVTBL` has five pointers to the following routine types:

```
int  (*svt_get)(SV* sv, MAGIC* mg);
int  (*svt_set)(SV* sv, MAGIC* mg);
U32  (*svt_len)(SV* sv, MAGIC* mg);
int  (*svt_clear)(SV* sv, MAGIC* mg);
int  (*svt_free)(SV* sv, MAGIC* mg);
```

This MGVTBL structure is set at compile-time in `perl.h` and there are currently 19 types (or 21 with overloading turned on). These different structures contain pointers to various routines that perform additional actions depending on which function is being called.

```
Function pointer    Action taken
----------------    ------------
svt_get             Do something after the value of the SV is retrieved.
svt_set             Do something after the SV is assigned a value.
svt_len             Report on the SV's length.
svt_clear           Clear something the SV represents.
svt_free            Free any extra storage associated with the SV.
```

For instance, the MGVTBL structure called `vtbl_sv` (which corresponds to an `mg_type` of '\0') contains:

```
{ magic_get, magic_set, magic_len, 0, 0 }
```

Thus, when an SV is determined to be magical and of type '\0', if a get operation is being performed, the routine `magic_get` is called. All the various routines for the various magical types begin with `magic_`.

The current kinds of Magic Virtual Tables are:

```
mg_type  MGVTBL              Type of magic
-------  ------              ----------------------------
\0       vtbl_sv             Special scalar variable
A        vtbl_amagic         %OVERLOAD hash
a        vtbl_amagicelem     %OVERLOAD hash element
c        (none)              Holds overload table (AMT) on stash
B        vtbl_bm             Boyer-Moore (fast string search)
E        vtbl_env            %ENV hash
e        vtbl_envelem        %ENV hash element
f        vtbl_fm             Formline ('compiled' format)
g        vtbl_mglob          m//g target / study()ed string
I        vtbl_isa            @ISA array
i        vtbl_isaelem        @ISA array element
k        vtbl_nkeys          scalar(keys()) lvalue
L        (none)              Debugger %_<filename
l        vtbl_dbline         Debugger %_<filename element
o        vtbl_collxfrm       Locale transformation
P        vtbl_pack           Tied array or hash
p        vtbl_packelem       Tied array or hash element
q        vtbl_packelem       Tied scalar or handle
S        vtbl_sig            %SIG hash
s        vtbl_sigelem        %SIG hash element
t        vtbl_taint          Taintedness
U        vtbl_uvar           Available for use by extensions
v        vtbl_vec            vec() lvalue
x        vtbl_substr         substr() lvalue
y        vtbl_defelem        Shadow "foreach" iterator variable /
                              smart parameter vivification
*        vtbl_glob           GV (typeglob)
#        vtbl_arylen         Array length ($#ary)
.        vtbl_pos            pos() lvalue
~        (none)              Available for use by extensions
```

When an uppercase and lowercase letter both exist in the table, then the uppercase letter is used to represent some kind of composite type (a list or a hash), and the lowercase letter is used to represent an element of that composite type.

The '~' and 'U' magic types are defined specifically for use by extensions and will not be used by perl itself. Extensions can use '~' magic to 'attach' private information to variables (typically objects). This is especially useful because there is no way for normal perl code to corrupt this private information (unlike using extra elements of a hash object).

Similarly, 'U' magic can be used much like *tie()* to call a C function any time a scalar's value is used or changed. The `MAGIC`'s `mg_ptr` field points to a `ufuncs` structure:

```
struct ufuncs {
    I32 (*uf_val)(IV, SV*);
    I32 (*uf_set)(IV, SV*);
    IV uf_index;
};
```

When the SV is read from or written to, the `uf_val` or `uf_set` function will be called with `uf_index` as the first arg and a pointer to the SV as the second.

Note that because multiple extensions may be using '~' or 'U' magic, it is important for extensions to take extra care to avoid conflict. Typically only using the magic on objects blessed into the same class as the extension is sufficient. For '~' magic, it may also be appropriate to add an I32 'signature' at the top of the private data area and check that.

Also note that the `sv_set*()` and `sv_cat*()` functions described earlier do **not** invoke 'set' magic on their targets. This must be done by the user either by calling the `SvSETMAGIC()` macro after calling these functions, or by using one of the `sv_set*_mg()` or `sv_cat*_mg()` functions. Similarly, generic C code must call the `SvGETMAGIC()` macro to invoke any 'get' magic if they use an SV obtained from external sources in functions that don't handle magic. the section on *API LISTING* later in this document identifies such functions. For example, calls to the `sv_cat*()` functions typically need to be followed by `SvSETMAGIC()`, but they don't need a prior `SvGETMAGIC()` since their implementation handles 'get' magic.

### Finding Magic

```
MAGIC* mg_find(SV*, int type); /* Finds the magic pointer of that type */
```

This routine returns a pointer to the `MAGIC` structure stored in the SV. If the SV does not have that magical feature, `NULL` is returned. Also, if the SV is not of type SVt_PVMG, Perl may core dump.

```
int mg_copy(SV* sv, SV* nsv, char* key, STRLEN klen);
```

This routine checks to see what types of magic `sv` has. If the mg_type field is an uppercase letter, then the mg_obj is copied to `nsv`, but the mg_type field is changed to be the lowercase letter.

### Understanding the Magic of Tied Hashes and Arrays

Tied hashes and arrays are magical beasts of the 'P' magic type.

WARNING: As of the 5.004 release, proper usage of the array and hash access functions requires understanding a few caveats. Some of these caveats are actually considered bugs in the API, to be fixed in later releases, and are bracketed with [MAYCHANGE] below. If you find yourself actually applying such information in this section, be aware that the behavior may change in the future, umm, without warning.

The `av_store` function, when given a tied array argument, merely copies the magic of the array onto the value to be "stored", using `mg_copy`. It may also return NULL, indicating that the value did not actually need to be stored in the array. [MAYCHANGE] After a call to `av_store` on a tied array, the caller will usually need to call `mg_set(val)` to actually invoke the perl level "STORE" method on the TIEARRAY object. If `av_store` did return NULL, a call to `SvREFCNT_dec(val)` will also be usually necessary to avoid a memory leak. [/MAYCHANGE]

The previous paragraph is applicable verbatim to tied hash access using the `hv_store` and `hv_store_ent` functions as well.

`av_fetch` and the corresponding hash functions `hv_fetch` and `hv_fetch_ent` actually return an undefined mortal value whose magic has been initialized using `mg_copy`. Note the value so returned does not need to be deallocated, as it is already mortal. [MAYCHANGE] But you will need to call `mg_get()` on the returned value in order to actually invoke the perl level "FETCH" method on the underlying TIE object. Similarly, you may also call `mg_set()` on the return value after possibly assigning a suitable value to it using `sv_setsv`, which will invoke the "STORE" method on the TIE object. [/MAYCHANGE]

[MAYCHANGE] In other words, the array or hash fetch/store functions don't really fetch and store actual values in the case of tied arrays and hashes. They merely call `mg_copy` to attach magic to the values that were meant to be "stored" or "fetched". Later calls to `mg_get` and `mg_set` actually do the job of invoking the TIE methods on the underlying objects. Thus the magic mechanism currently implements a kind of lazy access to arrays and hashes.

Currently (as of perl version 5.004), use of the hash and array access functions requires the user to be aware of whether they are operating on "normal" hashes and arrays, or on their tied variants. The API may be changed to provide more transparent access to both tied and normal data types in future versions. [/MAYCHANGE]

You would do well to understand that the TIEARRAY and TIEHASH interfaces are mere sugar to invoke some perl method calls while using the uniform hash and array syntax. The use of this sugar imposes some overhead (typically about two to four extra opcodes per FETCH/STORE operation, in addition to the creation of all the mortal variables required to invoke the methods). This overhead will be comparatively small if the TIE methods are themselves substantial, but if they are only a few statements long, the overhead will not be insignificant.

### Localizing changes

Perl has a very handy construction

```
{
  local $var = 2;
  ...
}
```

This construction is *approximately* equivalent to

```
{
  my $oldvar = $var;
  $var = 2;
  ...
  $var = $oldvar;
}
```

The biggest difference is that the first construction would reinstate the initial value of `$var`, irrespective of how control exits the block: `goto`, `return`, `die`/`eval` etc. It is a little bit more efficient as well.

There is a way to achieve a similar task from C via Perl API: create a *pseudo-block*, and arrange for some changes to be automatically undone at the end of it, either explicit, or via a non-local exit (via *die()*). A *block*–like construct is created by a pair of `ENTER`/`LEAVE` macros (see the section on *EXAMPLE/"Returning a Scalar* in the *perlcall* manpage). Such a construct may be created specially for some important localized task, or an existing one (like boundaries of enclosing Perl subroutine/block, or an existing pair for freeing TMPs) may be used. (In the second case the overhead of additional localization must be almost negligible.) Note that any XSUB is automatically enclosed in an `ENTER`/`LEAVE` pair.

Inside such a *pseudo-block* the following service is available:

`SAVEINT(int i)`

`SAVEIV(IV i)`

`SAVEI32(I32 i)`

`SAVELONG(long i)`
: These macros arrange things to restore the value of integer variable `i` at the end of enclosing *pseudo-block*.

`SAVESPTR(s)`

`SAVEPPTR(p)`
: These macros arrange things to restore the value of pointers `s` and `p`. `s` must be a pointer of a type which survives conversion to `SV*` and back, `p` should be able to survive conversion to `char*` and back.

`SAVEFREESV(SV *sv)`
: The refcount of `sv` would be decremented at the end of *pseudo-block*. This is similar to `sv_2mortal`, which should (?) be used instead.

`SAVEFREEOP(OP *op)`
: The `OP *` is *op_free()*ed at the end of *pseudo-block*.

`SAVEFREEPV(p)`
The chunk of memory which is pointed to by `p` is *Safefree()*ed at the end of *pseudo-block*.

`SAVECLEARSV(SV *sv)`
Clears a slot in the current scratchpad which corresponds to `sv` at the end of *pseudo-block*.

`SAVEDELETE(HV *hv, char *key, I32 length)`
The key `key` of `hv` is deleted at the end of *pseudo-block*. The string pointed to by `key` is *Safefree()*ed. If one has a *key* in short-lived storage, the corresponding string may be reallocated like this:

```
SAVEDELETE(PL_defstash, savepv(tmpbuf), strlen(tmpbuf));
```

`SAVEDESTRUCTOR(f,p)`
At the end of *pseudo-block* the function `f` is called with the only argument (of type `void*`) `p`.

`SAVESTACK_POS()`
The current offset on the Perl internal stack (cf. `SP`) is restored at the end of *pseudo-block*.

The following API list contains functions, thus one needs to provide pointers to the modifiable data explicitly (either C pointers, or Perlish `GV *`s). Where the above macros take `int`, a similar function takes `int *`.

`SV* save_scalar(GV *gv)`
Equivalent to Perl code `local $gv`.

`AV* save_ary(GV *gv)`

`HV* save_hash(GV *gv)`
Similar to `save_scalar`, but localize `@gv` and `%gv`.

`void save_item(SV *item)`
Duplicates the current value of `SV`, on the exit from the current `ENTER`/`LEAVE` *pseudo-block* will restore the value of `SV` using the stored value.

`void save_list(SV **sarg, I32 maxsarg)`
A variant of `save_item` which takes multiple arguments via an array `sarg` of `SV*` of length `maxsarg`.

`SV* save_svref(SV **sptr)`
Similar to `save_scalar`, but will reinstate a `SV *`.

`void save_aptr(AV **aptr)`

`void save_hptr(HV **hptr)`
Similar to `save_svref`, but localize `AV *` and `HV *`.

The `Alias` module implements localization of the basic types within the *caller's scope*. People who are interested in how to localize things in the containing scope should take a look there too.

## Subroutines

### XSUBs and the Argument Stack

The XSUB mechanism is a simple way for Perl programs to access C subroutines. An XSUB routine will have a stack that contains the arguments from the Perl program, and a way to map from the Perl data structures to a C equivalent.

The stack arguments are accessible through the `ST(n)` macro, which returns the `n`'th stack argument. Argument 0 is the first argument passed in the Perl subroutine call. These arguments are `SV*`, and can be used anywhere an `SV*` is used.

Most of the time, output from the C routine can be handled through use of the RETVAL and OUTPUT directives. However, there are some cases where the argument stack is not already long enough to handle all the return values. An example is the POSIX *tzname()* call, which takes no arguments, but returns two, the local time zone's standard and summer time abbreviations.

To handle this situation, the PPCODE directive is used and the stack is extended using the macro:

```
EXTEND(SP, num);
```

where `SP` is the macro that represents the local copy of the stack pointer, and `num` is the number of elements the stack should be extended by.

Now that there is room on the stack, values can be pushed on it using the macros to push IVs, doubles, strings, and SV pointers respectively:

```
PUSHi(IV)
PUSHn(double)
PUSHp(char*, I32)
PUSHs(SV*)
```

And now the Perl program calling `tzname`, the two values will be assigned as in:

```
($standard_abbrev, $summer_abbrev) = POSIX::tzname;
```

An alternate (and possibly simpler) method to pushing values on the stack is to use the macros:

```
XPUSHi(IV)
XPUSHn(double)
XPUSHp(char*, I32)
XPUSHs(SV*)
```

These macros automatically adjust the stack for you, if needed. Thus, you do not need to call `EXTEND` to extend the stack.

For more information, consult the *perlxs* manpage and the *perlxstut* manpage.

### Calling Perl Routines from within C Programs

There are four routines that can be used to call a Perl subroutine from within a C program. These four are:

```
I32  perl_call_sv(SV*, I32);
I32  perl_call_pv(char*, I32);
I32  perl_call_method(char*, I32);
I32  perl_call_argv(char*, I32, register char**);
```

The routine most often used is `perl_call_sv`. The `SV*` argument contains either the name of the Perl subroutine to be called, or a reference to the subroutine. The second argument consists of flags that control the context in which the subroutine is called, whether or not the subroutine is being passed arguments, how errors should be trapped, and how to treat return values.

All four routines return the number of arguments that the subroutine returned on the Perl stack.

When using any of these routines (except `perl_call_argv`), the programmer must manipulate the Perl stack. These include the following macros and functions:

```
dSP
SP
PUSHMARK()
PUTBACK
SPAGAIN
ENTER
SAVETMPS
FREETMPS
LEAVE
XPUSH*()
POP*()
```

For a detailed description of calling conventions from C to Perl, consult the *perlcall* manpage.

**Memory Allocation**

It is suggested that you use the version of malloc that is distributed with Perl. It keeps pools of various sizes of unallocated memory in order to satisfy allocation requests more quickly. However, on some platforms, it may cause spurious malloc or free errors.

```
New(x, pointer, number, type);
Newc(x, pointer, number, type, cast);
Newz(x, pointer, number, type);
```

These three macros are used to initially allocate memory.

The first argument `x` was a "magic cookie" that was used to keep track of who called the macro, to help when debugging memory problems. However, the current code makes no use of this feature (most Perl developers now use run-time memory checkers), so this argument can be any number.

The second argument `pointer` should be the name of a variable that will point to the newly allocated memory.

The third and fourth arguments `number` and `type` specify how many of the specified type of data structure should be allocated. The argument `type` is passed to `sizeof`. The final argument to `Newc`, `cast`, should be used if the `pointer` argument is different from the `type` argument.

Unlike the `New` and `Newc` macros, the `Newz` macro calls `memzero` to zero out all the newly allocated memory.

```
Renew(pointer, number, type);
Renewc(pointer, number, type, cast);
Safefree(pointer)
```

These three macros are used to change a memory buffer size or to free a piece of memory no longer needed. The arguments to `Renew` and `Renewc` match those of `New` and `Newc` with the exception of not needing the "magic cookie" argument.

```
Move(source, dest, number, type);
Copy(source, dest, number, type);
Zero(dest, number, type);
```

These three macros are used to move, copy, or zero out previously allocated memory. The `source` and `dest` arguments point to the source and destination starting points. Perl will move, copy, or zero out `number` instances of the size of the `type` data structure (using the `sizeof` function).

### PerlIO

The most recent development releases of Perl has been experimenting with removing Perl's dependency on the ''normal'' standard I/O suite and allowing other stdio implementations to be used. This involves creating a new abstraction layer that then calls whichever implementation of stdio Perl was compiled with. All XSUBs should now use the functions in the PerlIO abstraction layer and not make any assumptions about what kind of stdio is being used.

For a complete description of the PerlIO abstraction, consult the *perlapio* manpage.

### Putting a C value on Perl stack

A lot of opcodes (this is an elementary operation in the internal perl stack machine) put an `SV*` on the stack. However, as an optimization the corresponding SV is (usually) not recreated each time. The opcodes reuse specially assigned SVs (*target*s) which are (as a corollary) not constantly freed/created.

Each of the targets is created only once (but see the section on *Scratchpads and recursion* below), and when an opcode needs to put an integer, a double, or a string on stack, it just sets the corresponding parts of its *target* and puts the *target* on stack.

The macro to put this target on stack is `PUSHTARG`, and it is directly used in some opcodes, as well as indirectly in zillions of others, which use it via `(X)PUSH[pni]`.

### Scratchpads

The question remains on when the SVs which are *target*s for opcodes are created. The answer is that they are created when the current unit -- a subroutine or a file (for opcodes for statements outside of subroutines) -- is compiled. During this time a special anonymous Perl array is created, which is called a scratchpad for the current unit.

A scratchpad keeps SVs which are lexicals for the current unit and are targets for opcodes. One can deduce that an SV lives on a scratchpad by looking on its flags: lexicals have `SVs_PADMY` set, and *target*s have `SVs_PADTMP` set.

The correspondence between OPs and *target*s is not 1-to-1. Different OPs in the compile tree of the unit can use the same target, if this would not conflict with the expected life of the temporary.

### Scratchpads and recursion

In fact it is not 100% true that a compiled unit contains a pointer to the scratchpad AV. In fact it contains a pointer to an AV of (initially) one element, and this element is the scratchpad AV. Why do we need an extra level of indirection?

The answer is **recursion**, and maybe (sometime soon) **threads**. Both these can create several execution pointers going into the same subroutine. For the subroutine-child not write over the temporaries for the subroutine-parent (lifespan of which covers the call to the child), the parent and the child should have different scratchpads. (*And* the lexicals should be separate anyway!)

So each subroutine is born with an array of scratchpads (of length 1). On each entry to the subroutine it is checked that the current depth of the recursion is not more than the length of this array, and if it is, new scratchpad is created and pushed into the array.

The *target*s on this scratchpad are `undef`s, but they are already marked with correct flags.

## Compiled code

### Code tree

Here we describe the internal form your code is converted to by Perl. Start with a simple example:

```
$a = $b + $c;
```

This is converted to a tree similar to this one:

```
              assign-to
            /           \
           +             $a
         /   \
       $b     $c
```

(but slightly more complicated). This tree reflects the way Perl parsed your code, but has nothing to do with the execution order. There is an additional "thread" going through the nodes of the tree which shows the order of execution of the nodes. In our simplified example above it looks like:

```
$b ---> $c ---> + ---> $a ---> assign-to
```

But with the actual compile tree for `$a = $b + $c` it is different: some nodes *optimized away*. As a corollary, though the actual tree contains more nodes than our simplified example, the execution order is the same as in our example.

**Examining the tree**

If you have your perl compiled for debugging (usually done with `-D optimize=-g` on `Configure` command line), you may examine the compiled tree by specifying `-Dx` on the Perl command line. The output takes several lines per node, and for `$b+$c` it looks like this:

```
5           TYPE = add  ===> 6
            TARG = 1
            FLAGS = (SCALAR,KIDS)
            {
                TYPE = null  ===> (4)
                  (was rv2sv)
                FLAGS = (SCALAR,KIDS)
                {
3                   TYPE = gvsv  ===> 4
                    FLAGS = (SCALAR)
                    GV = main::b
                }
            }
            {
                TYPE = null  ===> (5)
                  (was rv2sv)
                FLAGS = (SCALAR,KIDS)
                {
4                   TYPE = gvsv  ===> 5
                    FLAGS = (SCALAR)
                    GV = main::c
                }
            }
```

This tree has 5 nodes (one per `TYPE` specifier), only 3 of them are not optimized away (one per number in the left column). The immediate children of the given node correspond to `{}` pairs on the same level of indentation, thus this listing corresponds to the tree:

```
       add
     /     \
   null    null
    |       |
   gvsv    gvsv
```

The execution order is indicated by ===> marks, thus it is `3 4 5 6` (node `6` is not included into above listing), i.e., `gvsv gvsv add whatever`.

### Compile pass 1: check routines

The tree is created by the *pseudo-compiler* while yacc code feeds it the constructions it recognizes. Since yacc works bottom-up, so does the first pass of perl compilation.

What makes this pass interesting for perl developers is that some optimization may be performed on this pass. This is optimization by so-called *check routines*. The correspondence between node names and corresponding check routines is described in *opcode.pl* (do not forget to run `make regen_headers` if you modify this file).

A check routine is called when the node is fully constructed except for the execution-order thread. Since at this time there are no back-links to the currently constructed node, one can do most any operation to the top-level node, including freeing it and/or creating new nodes above/below it.

The check routine returns the node which should be inserted into the tree (if the top-level node was not modified, check routine returns its argument).

By convention, check routines have names `ck_*`. They are usually called from `new*OP` subroutines (or `convert`) (which in turn are called from *perly.y*).

### Compile pass 1a: constant folding

Immediately after the check routine is called the returned node is checked for being compile-time executable. If it is (the value is judged to be constant) it is immediately executed, and a *constant* node with the ''return value'' of the corresponding subtree is substituted instead. The subtree is deleted.

If constant folding was not performed, the execution-order thread is created.

### Compile pass 2: context propagation

When a context for a part of compile tree is known, it is propagated down through the tree. At this time the context can have 5 values (instead of 2 for runtime context): void, boolean, scalar, list, and lvalue. In contrast with the pass 1 this pass is processed from top to bottom: a node's context determines the context for its children.

Additional context-dependent optimizations are performed at this time. Since at this moment the compile tree contains back-references (via ''thread'' pointers), nodes cannot be *free()*d now. To allow optimized-away nodes at this stage, such nodes are *null()*ified instead of *free()*ing (i.e. their type is changed to OP_NULL).

### Compile pass 3: peephole optimization

After the compile tree for a subroutine (or for an `eval` or a file) is created, an additional pass over the code is performed. This pass is neither top-down or bottom-up, but in the execution order (with additional complications for conditionals). These optimizations are done in the subroutine *peep()*. Optimizations performed at this stage are subject to the same restrictions as in the pass 2.

## API LISTING

This is a listing of functions, macros, flags, and variables that may be useful to extension writers or that may be found while reading other extensions.

Note that all Perl API global variables must be referenced with the `PL_` prefix. Some macros are provided for compatibility with the older, unadorned names, but this support will be removed in a future release.

It is strongly recommended that all Perl API functions that don't begin with `perl` be referenced with an explicit `Perl_` prefix.

The sort order of the listing is case insensitive, with any occurrences of '_' ignored for the the purpose of sorting.

av_clear Clears an array, making it empty. Does not free the memory used by the array itself.

```
void    av_clear (AV* ar)
```

av_extend
Pre-extend an array. The `key` is the index to which the array should be extended.

```
void    av_extend (AV* ar, I32 key)
```

av_fetch Returns the SV at the specified index in the array. The `key` is the index. If `lval` is set then the fetch will be part of a store. Check that the return value is non-null before dereferencing it to a `SV*`.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use this function on tied arrays.

```
SV**    av_fetch (AV* ar, I32 key, I32 lval)
```

AvFILL Same as `av_len()`. Deprecated, use `av_len()` instead.

av_len Returns the highest index in the array. Returns –1 if the array is empty.

```
I32     av_len (AV* ar)
```

av_make
Creates a new AV and populates it with a list of SVs. The SVs are copied into the array, so they may be freed after the call to av_make. The new AV will have a reference count of 1.

```
AV*     av_make (I32 size, SV** svp)
```

av_pop Pops an SV off the end of the array. Returns `&PL_sv_undef` if the array is empty.

```
SV*     av_pop (AV* ar)
```

av_push Pushes an SV onto the end of the array. The array will grow automatically to accommodate the addition.

```
void    av_push (AV* ar, SV* val)
```

av_shift Shifts an SV off the beginning of the array.

```
SV*     av_shift (AV* ar)
```

av_store Stores an SV in an array. The array index is specified as `key`. The return value will be NULL if the operation failed or if the value did not need to be actually stored within the array (as in the case of tied arrays). Otherwise it can be dereferenced to get the original `SV*`. Note that the caller is responsible for suitably incrementing the reference count of `val` before the call, and decrementing it if the function returned NULL.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use this function on tied arrays.

```
SV**    av_store (AV* ar, I32 key, SV* val)
```

av_undef
Undefines the array. Frees the memory used by the array itself.

```
void    av_undef (AV* ar)
```

av_unshift
Unshift the given number of `undef` values onto the beginning of the array. The array will grow automatically to accommodate the addition. You must then use `av_store` to assign values to these new elements.

```
void    av_unshift (AV* ar, I32 num)
```

CLASS
Variable which is setup by `xsubpp` to indicate the class name for a C++ XS constructor. This is always a `char*`. See `THIS` and the section on *Using XS With C++* in the *perlxs* manpage.

Copy
The XSUB−writer's interface to the C `memcpy` function. The `s` is the source, `d` is the destination, `n` is the number of items, and `t` is the type. May fail on overlapping copies. See also `Move`.

```
void    Copy( s, d, n, t )
```

croak
This is the XSUB−writer's interface to Perl's `die` function. Use this function the same way you use the C `printf` function. See `warn`.

CvSTASH
Returns the stash of the CV.

```
HV*     CvSTASH( SV* sv )
```

PL_DBsingle
When Perl is run in debugging mode, with the **−d** switch, this SV is a boolean which indicates whether subs are being single-stepped. Single-stepping is automatically turned on after every step. This is the C variable which corresponds to Perl's `$DB::single` variable. See `PL_DBsub`.

PL_DBsub
When Perl is run in debugging mode, with the **−d** switch, this GV contains the SV which holds the name of the sub being debugged. This is the C variable which corresponds to Perl's `$DB::sub` variable. See `PL_DBsingle`. The sub name can be found by

```
SvPV( GvSV( PL_DBsub ), PL_na )
```

PL_DBtrace
Trace variable used when Perl is run in debugging mode, with the **−d** switch. This is the C variable which corresponds to Perl's `$DB::trace` variable. See `PL_DBsingle`.

dMARK
Declare a stack marker variable, `mark`, for the XSUB. See `MARK` and `dORIGMARK`.

dORIGMARK
Saves the original stack mark for the XSUB. See `ORIGMARK`.

PL_dowarn
The C variable which corresponds to Perl's $^W warning variable.

dSP
Declares a local copy of perl's stack pointer for the XSUB, available via the `SP` macro. See `SP`.

dXSARGS
Sets up stack and mark pointers for an XSUB, calling dSP and dMARK. This is usually handled automatically by `xsubpp`. Declares the `items` variable to indicate the number of items on the stack.

dXSI32 Sets up the `ix` variable for an XSUB which has aliases. This is usually handled automatically by `xsubpp`.

do_binmode
Switches filehandle to binmode. `iotype` is what `IoTYPE(io)` would contain.

```
do_binmode(fp, iotype, TRUE);
```

ENTER Opening bracket on a callback. See `LEAVE` and the *perlcall* manpage.

```
ENTER;
```

EXTEND
Used to extend the argument stack for an XSUB's return values.

```
EXTEND( sp, int x )
```

fbm_compile
Analyses the string in order to make fast searches on it using *fbm_instr()* -- the Boyer-Moore algorithm.

```
void    fbm_compile(SV* sv, U32 flags)
```

fbm_instr
Returns the location of the SV in the string delimited by `str` and `strend`. It returns `Nullch` if the string can't be found. The `sv` does not have to be fbm_compiled, but the search will not be as fast then.

```
char*   fbm_instr(char *str, char *strend, SV *sv, U32 flags)
```

FREETMPS
Closing bracket for temporaries on a callback. See `SAVETMPS` and the *perlcall* manpage.

```
FREETMPS;
```

G_ARRAY
Used to indicate array context. See `GIMME_V`, `GIMME` and the *perlcall* manpage.

G_DISCARD
Indicates that arguments returned from a callback should be discarded. See the *perlcall* manpage.

G_EVAL
Used to force a Perl `eval` wrapper around a callback. See the *perlcall* manpage.

GIMME A backward-compatible version of `GIMME_V` which can only return `G_SCALAR` or `G_ARRAY`; in a void context, it returns `G_SCALAR`.

GIMME_V
The XSUB−writer's equivalent to Perl's `wantarray`. Returns `G_VOID`, `G_SCALAR` or `G_ARRAY` for void, scalar or array context, respectively.

G_NOARGS
Indicates that no arguments are being sent to a callback. See the *perlcall* manpage.

G_SCALAR
Used to indicate scalar context. See `GIMME_V`, `GIMME`, and the *perlcall* manpage.

gv_fetchmeth

Returns the glob with the given `name` and a defined subroutine or `NULL`. The glob lives in the given `stash`, or in the stashes accessible via `@ISA` and `@UNIVERSAL`.

The argument `level` should be either 0 or −1. If `level==0`, as a side-effect creates a glob with the given `name` in the given `stash` which in the case of success contains an alias for the subroutine, and sets up caching info for this glob. Similarly for all the searched stashes.

This function grants `"SUPER"` token as a postfix of the stash name.

The GV returned from `gv_fetchmeth` may be a method cache entry, which is not visible to Perl code. So when calling `perl_call_sv`, you should not use the GV directly; instead, you should use the method's CV, which can be obtained from the GV with the `GvCV` macro.

```
GV*     gv_fetchmeth (HV* stash, char* name, STRLEN len, I32 level)
```

gv_fetchmethod

gv_fetchmethod_autoload

Returns the glob which contains the subroutine to call to invoke the method on the `stash`. In fact in the presense of autoloading this may be the glob for "AUTOLOAD". In this case the corresponding variable `$AUTOLOAD` is already setup.

The third parameter of `gv_fetchmethod_autoload` determines whether AUTOLOAD lookup is performed if the given method is not present: non-zero means yes, look for AUTOLOAD; zero means no, don't look for AUTOLOAD. Calling `gv_fetchmethod` is equivalent to calling `gv_fetchmethod_autoload` with a non-zero `autoload` parameter.

These functions grant `"SUPER"` token as a prefix of the method name.

Note that if you want to keep the returned glob for a long time, you need to check for it being "AUTOLOAD", since at the later time the call may load a different subroutine due to `$AUTOLOAD` changing its value. Use the glob created via a side effect to do this.

These functions have the same side-effects and as `gv_fetchmeth` with `level==0`. `name` should be writable if contains `':'` or `'\''`. The warning against passing the GV returned by `gv_fetchmeth` to `perl_call_sv` apply equally to these functions.

```
GV*     gv_fetchmethod (HV* stash, char* name)
GV*     gv_fetchmethod_autoload (HV* stash, char* name, I32 autoload)
```

G_VOID

Used to indicate void context. See `GIMME_V` and the *perlcall* manpage.

gv_stashpv

Returns a pointer to the stash for a specified package. If `create` is set then the package will be created if it does not already exist. If `create` is not set and the package does not exist then NULL is returned.

```
HV*     gv_stashpv (char* name, I32 create)
```

gv_stashsv

Returns a pointer to the stash for a specified package. See `gv_stashpv`.

```
HV*     gv_stashsv (SV* sv, I32 create)
```

GvSV Return the SV from the GV.

HEf_SVKEY
This flag, used in the length slot of hash entries and magic structures, specifies the structure contains a `SV*` pointer where a `char*` pointer is to be expected. (For information only— not to be used).

HeHASH
Returns the computed hash stored in the hash entry.

```
U32     HeHASH(HE* he)
```

HeKEY
Returns the actual pointer stored in the key slot of the hash entry. The pointer may be either `char*` or `SV*`, depending on the value of `HeKLEN()`. Can be assigned to. The `HePV()` or `HeSVKEY()` macros are usually preferable for finding the value of a key.

```
char*   HeKEY(HE* he)
```

HeKLEN
If this is negative, and amounts to `HEf_SVKEY`, it indicates the entry holds an `SV*` key. Otherwise, holds the actual length of the key. Can be assigned to. The `HePV()` macro is usually preferable for finding key lengths.

```
int     HeKLEN(HE* he)
```

HePV
Returns the key slot of the hash entry as a `char*` value, doing any necessary dereferencing of possibly `SV*` keys. The length of the string is placed in `len` (this is a macro, so do *not* use `&len`). If you do not care about what the length of the key is, you may use the global variable `PL_na`. Remember though, that hash keys in perl are free to contain embedded nulls, so using `strlen()` or similar is not a good way to find the length of hash keys. This is very similar to the `SvPV()` macro described elsewhere in this document.

```
char*   HePV(HE* he, STRLEN len)
```

HeSVKEY
Returns the key as an `SV*`, or `Nullsv` if the hash entry does not contain an `SV*` key.

```
HeSVKEY(HE* he)
```

HeSVKEY_force
Returns the key as an `SV*`. Will create and return a temporary mortal `SV*` if the hash entry contains only a `char*` key.

```
HeSVKEY_force(HE* he)
```

HeSVKEY_set
Sets the key to a given `SV*`, taking care to set the appropriate flags to indicate the presence of an `SV*` key, and returns the same `SV*`.

```
HeSVKEY_set(HE* he, SV* sv)
```

HeVAL
Returns the value slot (type `SV*`) stored in the hash entry.

```
HeVAL(HE* he)
```

hv_clear Clears a hash, making it empty.

```
void    hv_clear (HV* tb)
```

hv_delayfree_ent
Releases a hash entry, such as while iterating though the hash, but delays actual freeing of key and value until the end of the current statement (or thereabouts) with `sv_2mortal`. See `hv_iternext` and `hv_free_ent`.

```
void    hv_delayfree_ent (HV* hv, HE* entry)
```

hv_delete
Deletes a key/value pair in the hash. The value SV is removed from the hash and returned to the caller. The `klen` is the length of the key. The `flags` value will normally be zero; if set to G_DISCARD then NULL will be returned.

```
SV*     hv_delete (HV* tb, char* key, U32 klen, I32 flags)
```

hv_delete_ent
Deletes a key/value pair in the hash. The value SV is removed from the hash and returned to the caller. The `flags` value will normally be zero; if set to G_DISCARD then NULL will be returned. `hash` can be a valid precomputed hash value, or 0 to ask for it to be computed.

```
SV*     hv_delete_ent (HV* tb, SV* key, I32 flags, U32 hash)
```

hv_exists
Returns a boolean indicating whether the specified hash key exists. The `klen` is the length of the key.

```
bool    hv_exists (HV* tb, char* key, U32 klen)
```

hv_exists_ent
Returns a boolean indicating whether the specified hash key exists. `hash` can be a valid precomputed hash value, or 0 to ask for it to be computed.

```
bool    hv_exists_ent (HV* tb, SV* key, U32 hash)
```

hv_fetch Returns the SV which corresponds to the specified key in the hash. The `klen` is the length of the key. If `lval` is set then the fetch will be part of a store. Check that the return value is non-null before dereferencing it to a `SV*`.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use this function on tied hashes.

```
SV**    hv_fetch (HV* tb, char* key, U32 klen, I32 lval)
```

hv_fetch_ent
Returns the hash entry which corresponds to the specified key in the hash. `hash` must be a valid precomputed hash number for the given `key`, or 0 if you want the function to compute it. IF `lval` is set then the fetch will be part of a store. Make sure the return value is non-null before accessing it. The return value when `tb` is a tied hash is a pointer to a static location, so be sure to make a copy of the structure if you need to store it somewhere.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use this function on tied hashes.

```
HE*     hv_fetch_ent  (HV* tb, SV* key, I32 lval, U32 hash)
```

hv_free_ent

Releases a hash entry, such as while iterating though the hash. See `hv_iternext` and `hv_delayfree_ent`.

```
void    hv_free_ent (HV* hv, HE* entry)
```

hv_iterinit

Prepares a starting point to traverse a hash table.

```
I32     hv_iterinit (HV* tb)
```

Returns the number of keys in the hash (i.e. the same as `HvKEYS(tb)`). The return value is currently only meaningful for hashes without tie magic.

NOTE: Before version 5.004_65, `hv_iterinit` used to return the number of hash buckets that happen to be in use. If you still need that esoteric value, you can get it through the macro `HvFILL(tb)`.

hv_iterkey

Returns the key from the current position of the hash iterator. See `hv_iterinit`.

```
char*   hv_iterkey (HE* entry, I32* retlen)
```

hv_iterkeysv

Returns the key as an `SV*` from the current position of the hash iterator. The return value will always be a mortal copy of the key. Also see `hv_iterinit`.

```
SV*     hv_iterkeysv  (HE* entry)
```

hv_iternext

Returns entries from a hash iterator. See `hv_iterinit`.

```
HE*     hv_iternext (HV* tb)
```

hv_iternextsv

Performs an `hv_iternext`, `hv_iterkey`, and `hv_iterval` in one operation.

```
SV*     hv_iternextsv (HV* hv, char** key, I32* retlen)
```

hv_iterval

Returns the value from the current position of the hash iterator. See `hv_iterkey`.

```
SV*     hv_iterval (HV* tb, HE* entry)
```

hv_magic

Adds magic to a hash. See `sv_magic`.

```
void    hv_magic (HV* hv, GV* gv, int how)
```

HvNAME

Returns the package name of a stash. See `SvSTASH`, `CvSTASH`.

```
char*   HvNAME (HV* stash)
```

hv_store Stores an SV in a hash. The hash key is specified as `key` and `klen` is the length of the key. The `hash` parameter is the precomputed hash value; if it is zero then Perl will compute it. The return value will be NULL if the operation failed or if the value did not need to be actually stored within the hash (as in the case of tied hashes). Otherwise it can be dereferenced to get the original `SV*`. Note that the caller is responsible for suitably incrementing the reference count of `val` before the call, and decrementing it if the function returned NULL.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use this function on tied hashes.

```
SV**    hv_store (HV* tb, char* key, U32 klen, SV* val, U32 hash)
```

hv_store_ent

Stores `val` in a hash. The hash key is specified as `key`. The `hash` parameter is the precomputed hash value; if it is zero then Perl will compute it. The return value is the new hash entry so created. It will be NULL if the operation failed or if the value did not need to be actually stored within the hash (as in the case of tied hashes). Otherwise the contents of the return value can be accessed using the `He???` macros described here. Note that the caller is responsible for suitably incrementing the reference count of `val` before the call, and decrementing it if the function returned NULL.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use this function on tied hashes.

```
HE*     hv_store_ent  (HV* tb, SV* key, SV* val, U32 hash)
```

hv_undef

Undefines the hash.

```
void    hv_undef (HV* tb)
```

isALNUM

Returns a boolean indicating whether the C `char` is an ascii alphanumeric character or digit.

```
int     isALNUM (char c)
```

isALPHA

Returns a boolean indicating whether the C `char` is an ascii alphabetic character.

```
int     isALPHA (char c)
```

isDIGIT Returns a boolean indicating whether the C `char` is an ascii digit.

```
int     isDIGIT (char c)
```

isLOWER

Returns a boolean indicating whether the C `char` is a lowercase character.

```
int     isLOWER (char c)
```

isSPACE

Returns a boolean indicating whether the C `char` is whitespace.

```
int     isSPACE (char c)
```

isUPPER
Returns a boolean indicating whether the C `char` is an uppercase character.

```
int     isUPPER (char c)
```

items Variable which is setup by `xsubpp` to indicate the number of items on the stack. See the section on *Variable-length Parameter Lists* in the *perlxs* manpage.

ix Variable which is setup by `xsubpp` to indicate which of an XSUB's aliases was used to invoke it. See the section on *The ALIAS: Keyword* in the *perlxs* manpage.

LEAVE Closing bracket on a callback. See `ENTER` and the *perlcall* manpage.

```
LEAVE;
```

looks_like_number
Test if an the content of an SV looks like a number (or is a number).

```
int     looks_like_number(SV*)
```

MARK Stack marker variable for the XSUB. See `dMARK`.

mg_clear
Clear something magical that the SV represents. See `sv_magic`.

```
int     mg_clear (SV* sv)
```

mg_copy
Copies the magic from one SV to another. See `sv_magic`.

```
int     mg_copy (SV *, SV *, char *, STRLEN)
```

mg_find Finds the magic pointer for type matching the SV. See `sv_magic`.

```
MAGIC*  mg_find (SV* sv, int type)
```

mg_free Free any magic storage used by the SV. See `sv_magic`.

```
int     mg_free (SV* sv)
```

mg_get Do magic after a value is retrieved from the SV. See `sv_magic`.

```
int     mg_get (SV* sv)
```

mg_len Report on the SV's length. See `sv_magic`.

```
U32     mg_len (SV* sv)
```

mg_magical
Turns on the magical status of an SV. See `sv_magic`.

```
void    mg_magical (SV* sv)
```

mg_set Do magic after a value is assigned to the SV. See `sv_magic`.

```
int     mg_set (SV* sv)
```

**Move** The XSUB−writer's interface to the C `memmove` function. The `s` is the source, `d` is the destination, `n` is the number of items, and `t` is the type. Can do overlapping moves. See also `Copy`.

```
void    Move( s, d, n, t )
```

**PL_na** A variable which may be used with `SvPV` to tell Perl to calculate the string length.

**New** The XSUB−writer's interface to the C `malloc` function.

```
void*   New( x, void *ptr, int size, type )
```

**newAV** Creates a new AV. The reference count is set to 1.

```
AV*     newAV (void)
```

**Newc** The XSUB−writer's interface to the C `malloc` function, with cast.

```
void*   Newc( x, void *ptr, int size, type, cast )
```

**newCONSTSUB**
Creates a constant sub equivalent to Perl `sub FOO () { 123 }` which is eligible for inlining at compile-time.

```
void    newCONSTSUB(HV* stash, char* name, SV* sv)
```

**newHV** Creates a new HV. The reference count is set to 1.

```
HV*     newHV (void)
```

**newRV_inc**
Creates an RV wrapper for an SV. The reference count for the original SV is incremented.

```
SV*     newRV_inc (SV* ref)
```

For historical reasons, "newRV" is a synonym for "newRV_inc".

**newRV_noinc**
Creates an RV wrapper for an SV. The reference count for the original SV is **not** incremented.

```
SV*     newRV_noinc (SV* ref)
```

**NEWSV** Creates a new SV. A non-zero `len` parameter indicates the number of bytes of preallocated string space the SV should have. An extra byte for a tailing NUL is also reserved. (SvPOK is not set for the SV even if string space is allocated.) The reference count for the new SV is set to 1. `id` is an integer id between 0 and 1299 (used to identify leaks).

```
SV*     NEWSV (int id, STRLEN len)
```

**newSViv**
Creates a new SV and copies an integer into it. The reference count for the SV is set to 1.

```
SV*     newSViv (IV i)
```

**newSVnv**
Creates a new SV and copies a double into it. The reference count for the SV is set to 1.

```
SV*     newSVnv (NV i)
```

newSVpv
: Creates a new SV and copies a string into it. The reference count for the SV is set to 1. If `len` is zero then Perl will compute the length.

```
SV*     newSVpv (char* s, STRLEN len)
```

newSVpvf
: Creates a new SV an initialize it with the string formatted like `sprintf`.

```
SV*     newSVpvf(const char* pat, ...);
```

newSVpvn
: Creates a new SV and copies a string into it. The reference count for the SV is set to 1. If `len` is zero then Perl will create a zero length string.

```
SV*     newSVpvn (char* s, STRLEN len)
```

newSVrv
: Creates a new SV for the RV, `rv`, to point to. If `rv` is not an RV then it will be upgraded to one. If `classname` is non-null then the new SV will be blessed in the specified package. The new SV is returned and its reference count is 1.

```
SV*     newSVrv (SV* rv, char* classname)
```

newSVsv
: Creates a new SV which is an exact duplicate of the original SV.

```
SV*     newSVsv (SV* old)
```

newXS
: Used by `xsubpp` to hook up XSUBs as Perl subs.

newXSproto
: Used by `xsubpp` to hook up XSUBs as Perl subs. Adds Perl prototypes to the subs.

Newz
: The XSUB−writer's interface to the C `malloc` function. The allocated memory is zeroed with `memzero`.

```
void*   Newz( x, void *ptr, int size, type )
```

Nullav
: Null AV pointer.

Nullch
: Null character pointer.

Nullcv
: Null CV pointer.

Nullhv
: Null HV pointer.

Nullsv
: Null SV pointer.

ORIGMARK
: The original stack mark for the XSUB. See `dORIGMARK`.

perl_alloc
: Allocates a new Perl interpreter. See the *perlembed* manpage.

perl_call_argv
: Performs a callback to the specified Perl sub. See the *perlcall* manpage.

```
I32     perl_call_argv (char* subname, I32 flags, char** argv)
```

perl_call_method
: Performs a callback to the specified Perl method. The blessed object must be on the stack. See the *perlcall* manpage.

```
I32     perl_call_method (char* methname, I32 flags)
```

perl_call_pv
: Performs a callback to the specified Perl sub. See the *perlcall* manpage.

```
I32     perl_call_pv (char* subname, I32 flags)
```

perl_call_sv
: Performs a callback to the Perl sub whose name is in the SV. See the *perlcall* manpage.

```
I32     perl_call_sv (SV* sv, I32 flags)
```

perl_construct
: Initializes a new Perl interpreter. See the *perlembed* manpage.

perl_destruct
: Shuts down a Perl interpreter. See the *perlembed* manpage.

perl_eval_sv
: Tells Perl to `eval` the string in the SV.

```
I32     perl_eval_sv (SV* sv, I32 flags)
```

perl_eval_pv
: Tells Perl to `eval` the given string and return an SV* result.

```
SV*     perl_eval_pv (char* p, I32 croak_on_error)
```

perl_free
: Releases a Perl interpreter. See the *perlembed* manpage.

perl_get_av
: Returns the AV of the specified Perl array. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

```
AV*     perl_get_av (char* name, I32 create)
```

perl_get_cv
: Returns the CV of the specified Perl sub. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

```
CV*     perl_get_cv (char* name, I32 create)
```

perl_get_hv
: Returns the HV of the specified Perl hash. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

```
HV*     perl_get_hv (char* name, I32 create)
```

perl_get_sv
: Returns the SV of the specified Perl scalar. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

```
SV*     perl_get_sv (char* name, I32 create)
```

perl_parse
: Tells a Perl interpreter to parse a Perl script. See the *perlembed* manpage.

perl_require_pv
: Tells Perl to `require` a module.

```
void    perl_require_pv (char* pv)
```

perl_run
: Tells a Perl interpreter to run. See the *perlembed* manpage.

POPi
: Pops an integer off the stack.

```
int     POPi()
```

POPl
: Pops a long off the stack.

```
long    POPl()
```

POPp
: Pops a string off the stack.

```
char*   POPp()
```

POPn
: Pops a double off the stack.

```
double  POPn()
```

POPs
: Pops an SV off the stack.

```
SV*     POPs()
```

PUSHMARK
: Opening bracket for arguments on a callback. See `PUTBACK` and the *perlcall* manpage.

```
PUSHMARK(p)
```

PUSHi
: Push an integer onto the stack. The stack must have room for this element. Handles 'set' magic. See `XPUSHi`.

```
void    PUSHi(int d)
```

PUSHn
: Push a double onto the stack. The stack must have room for this element. Handles 'set' magic. See `XPUSHn`.

```
void    PUSHn(double d)
```

PUSHp
: Push a string onto the stack. The stack must have room for this element. The `len` indicates the length of the string. Handles 'set' magic. See `XPUSHp`.

```
void    PUSHp(char *c, int len )
```

PUSHs Push an SV onto the stack. The stack must have room for this element. Does not handle 'set' magic. See `XPUSHs`.

```
void    PUSHs(sv)
```

PUSHu Push an unsigned integer onto the stack. The stack must have room for this element. See `XPUSHu`.

```
void    PUSHu(unsigned int d)
```

PUTBACK
Closing bracket for XSUB arguments. This is usually handled by `xsubpp`. See `PUSHMARK` and the *perlcall* manpage for other uses.

```
PUTBACK;
```

Renew The XSUB−writer's interface to the C `realloc` function.

```
void*   Renew( void *ptr, int size, type )
```

Renewc The XSUB−writer's interface to the C `realloc` function, with cast.

```
void*   Renewc( void *ptr, int size, type, cast )
```

RETVAL Variable which is setup by `xsubpp` to hold the return value for an XSUB. This is always the proper type for the XSUB. See the section on *The RETVAL Variable* in the *perlxs* manpage.

safefree The XSUB−writer's interface to the C `free` function.

safemalloc
The XSUB−writer's interface to the C `malloc` function.

saferealloc
The XSUB−writer's interface to the C `realloc` function.

savepv Copy a string to a safe spot. This does not use an SV.

```
char*   savepv (char* sv)
```

savepvn Copy a string to a safe spot. The `len` indicates number of bytes to copy. This does not use an SV.

```
char*   savepvn (char* sv, I32 len)
```

SAVETMPS
Opening bracket for temporaries on a callback. See `FREETMPS` and the *perlcall* manpage.

```
SAVETMPS;
```

SP Stack pointer. This is usually handled by `xsubpp`. See `dSP` and `SPAGAIN`.

SPAGAIN
Refetch the stack pointer. Used after a callback. See the *perlcall* manpage.

```
SPAGAIN;
```

ST Used to access elements on the XSUB's stack.

```
SV*     ST(int x)
```

strEQ Test two strings to see if they are equal. Returns true or false.

```
int     strEQ( char *s1, char *s2 )
```

strGE Test two strings to see if the first, `s1`, is greater than or equal to the second, `s2`. Returns true or false.

```
int     strGE( char *s1, char *s2 )
```

strGT Test two strings to see if the first, `s1`, is greater than the second, `s2`. Returns true or false.

```
int     strGT( char *s1, char *s2 )
```

strLE Test two strings to see if the first, `s1`, is less than or equal to the second, `s2`. Returns true or false.

```
int     strLE( char *s1, char *s2 )
```

strLT Test two strings to see if the first, `s1`, is less than the second, `s2`. Returns true or false.

```
int     strLT( char *s1, char *s2 )
```

strNE Test two strings to see if they are different. Returns true or false.

```
int     strNE( char *s1, char *s2 )
```

strnEQ Test two strings to see if they are equal. The `len` parameter indicates the number of bytes to compare. Returns true or false.

```
int     strnEQ( char *s1, char *s2 )
```

strnNE Test two strings to see if they are different. The `len` parameter indicates the number of bytes to compare. Returns true or false.

```
int     strnNE( char *s1, char *s2, int len )
```

sv_2mortal
Marks an SV as mortal. The SV will be destroyed when the current context ends.

```
SV*     sv_2mortal (SV* sv)
```

sv_bless Blesses an SV into a specified package. The SV must be an RV. The package must be designated by its stash (see `gv_stashpv()`). The reference count of the SV is unaffected.

```
SV*     sv_bless (SV* sv, HV* stash)
```

sv_catpv
Concatenates the string onto the end of the string which is in the SV. Handles 'get' magic, but not 'set' magic. See `sv_catpv_mg`.

```
void    sv_catpv (SV* sv, char* ptr)
```

sv_catpv_mg
Like `sv_catpv`, but also handles 'set' magic.

```
void    sv_catpvn (SV* sv, char* ptr)
```

sv_catpvn
Concatenates the string onto the end of the string which is in the SV. The `len` indicates number of bytes to copy. Handles 'get' magic, but not 'set' magic. See `sv_catpvn_mg`.

```
void    sv_catpvn (SV* sv, char* ptr, STRLEN len)
```

sv_catpvn_mg
Like `sv_catpvn`, but also handles 'set' magic.

```
void    sv_catpvn_mg (SV* sv, char* ptr, STRLEN len)
```

sv_catpvf
Processes its arguments like `sprintf` and appends the formatted output to an SV. Handles 'get' magic, but not 'set' magic. `SvSETMAGIC()` must typically be called after calling this function to handle 'set' magic.

```
void    sv_catpvf (SV* sv, const char* pat, ...)
```

sv_catpvf_mg
Like `sv_catpvf`, but also handles 'set' magic.

```
void    sv_catpvf_mg (SV* sv, const char* pat, ...)
```

sv_catsv Concatenates the string from SV `ssv` onto the end of the string in SV `dsv`. Handles 'get' magic, but not 'set' magic. See `sv_catsv_mg`.

```
void    sv_catsv (SV* dsv, SV* ssv)
```

sv_catsv_mg
Like `sv_catsv`, but also handles 'set' magic.

```
void    sv_catsv_mg (SV* dsv, SV* ssv)
```

sv_chop Efficient removal of characters from the beginning of the string buffer. *SvPOK*(sv) must be true and the `ptr` must be a pointer to somewhere inside the string buffer. The `ptr` becomes the first character of the adjusted string.

```
void    sv_chop(SV* sv, char *ptr)
```

sv_cmp Compares the strings in two SVs. Returns −1, 0, or 1 indicating whether the string in `sv1` is less than, equal to, or greater than the string in `sv2`.

```
I32     sv_cmp (SV* sv1, SV* sv2)
```

SvCUR Returns the length of the string which is in the SV. See `SvLEN`.

```
int     SvCUR (SV* sv)
```

SvCUR_set
: Set the length of the string which is in the SV. See `SvCUR`.

```
void    SvCUR_set (SV* sv, int val )
```

sv_dec
: Auto-decrement of the value in the SV.

```
void    sv_dec (SV* sv)
```

sv_derived_from
: Returns a boolean indicating whether the SV is a subclass of the specified class.

```
int     sv_derived_from(SV* sv, char* class)
```

sv_derived_from
: Returns a boolean indicating whether the SV is derived from the specified class. This is the function that implements `UNIVERSAL::isa`. It works for class names as well as for objects.

```
bool    sv_derived_from _((SV* sv, char* name));
```

SvEND
: Returns a pointer to the last character in the string which is in the SV. See `SvCUR`. Access the character as

```
char*   SvEND(sv)
```

sv_eq
: Returns a boolean indicating whether the strings in the two SVs are identical.

```
I32     sv_eq (SV* sv1, SV* sv2)
```

SvGETMAGIC
: Invokes `mg_get` on an SV if it has 'get' magic. This macro evaluates its argument more than once.

```
void    SvGETMAGIC( SV *sv )
```

SvGROW
: Expands the character buffer in the SV so that it has room for the indicated number of bytes (remember to reserve space for an extra trailing NUL character). Calls `sv_grow` to perform the expansion if necessary. Returns a pointer to the character buffer.

```
char*   SvGROW( SV* sv, int len )
```

sv_grow
: Expands the character buffer in the SV. This will use `sv_unref` and will upgrade the SV to `SVt_PV`. Returns a pointer to the character buffer. Use `SvGROW`.

sv_inc
: Auto-increment of the value in the SV.

```
void    sv_inc (SV* sv)
```

sv_insert
: Inserts a string at the specified offset/length within the SV. Similar to the Perl *substr()* function.

```
void    sv_insert(SV *sv, STRLEN offset, STRLEN len,
                  char *str, STRLEN strlen)
```

SvIOK Returns a boolean indicating whether the SV contains an integer.

```
int     SvIOK (SV* SV)
```

SvIOK_off
Unsets the IV status of an SV.

```
void    SvIOK_off (SV* sv)
```

SvIOK_on
Tells an SV that it is an integer.

```
void    SvIOK_on (SV* sv)
```

SvIOK_only
Tells an SV that it is an integer and disables all other OK bits.

```
void    SvIOK_only (SV* sv)
```

SvIOKp Returns a boolean indicating whether the SV contains an integer. Checks the **private** setting. Use `SvIOK`.

```
int     SvIOKp (SV* SV)
```

sv_isa Returns a boolean indicating whether the SV is blessed into the specified class. This does not check for subtypes; use `sv_derived_from` to verify an inheritance relationship.

```
int     sv_isa (SV* sv, char* name)
```

sv_isobject
Returns a boolean indicating whether the SV is an RV pointing to a blessed object. If the SV is not an RV, or if the object is not blessed, then this will return false.

```
int     sv_isobject (SV* sv)
```

SvIV Returns the integer which is in the SV.

```
int SvIV (SV* sv)
```

SvIVX Returns the integer which is stored in the SV.

```
int     SvIVX (SV* sv)
```

SvLEN Returns the size of the string buffer in the SV. See `SvCUR`.

```
int     SvLEN (SV* sv)
```

sv_len Returns the length of the string in the SV. Use `SvCUR`.

```
STRLEN  sv_len (SV* sv)
```

sv_magic
Adds magic to an SV.

```
void    sv_magic (SV* sv, SV* obj, int how, char* name, I32 namlen)
```

sv_mortalcopy
Creates a new SV which is a copy of the original SV. The new SV is marked as mortal.

```
SV*     sv_mortalcopy (SV* oldsv)
```

sv_newmortal
Creates a new SV which is mortal. The reference count of the SV is set to 1.

```
SV*     sv_newmortal (void)
```

SvNIOK
Returns a boolean indicating whether the SV contains a number, integer or double.

```
int     SvNIOK (SV* SV)
```

SvNIOK_off
Unsets the NV/IV status of an SV.

```
void    SvNIOK_off (SV* sv)
```

SvNIOKp
Returns a boolean indicating whether the SV contains a number, integer or double. Checks the **private** setting. Use `SvNIOK`.

```
int     SvNIOKp (SV* SV)
```

PL_sv_no
This is the `false` SV. See `PL_sv_yes`. Always refer to this as `&PL_sv_no`.

SvNOK Returns a boolean indicating whether the SV contains a double.

```
int     SvNOK (SV* SV)
```

SvNOK_off
Unsets the NV status of an SV.

```
void    SvNOK_off (SV* sv)
```

SvNOK_on
Tells an SV that it is a double.

```
void    SvNOK_on (SV* sv)
```

SvNOK_only
Tells an SV that it is a double and disables all other OK bits.

```
void    SvNOK_only (SV* sv)
```

SvNOKp
Returns a boolean indicating whether the SV contains a double. Checks the **private** setting. Use `SvNOK`.

```
int     SvNOKp (SV* SV)
```

SvNV Returns the double which is stored in the SV.

```
double  SvNV (SV* sv)
```

SvNVX Returns the double which is stored in the SV.

```
double  SvNVX (SV* sv)
```

SvOK Returns a boolean indicating whether the value is an SV.

```
int     SvOK (SV* sv)
```

SvOOK Returns a boolean indicating whether the SvIVX is a valid offset value for the SvPVX. This hack is used internally to speed up removal of characters from the beginning of a SvPV. When SvOOK is true, then the start of the allocated string buffer is really (SvPVX − SvIVX).

```
int     SvOOK(SV* sv)
```

SvPOK Returns a boolean indicating whether the SV contains a character string.

```
int     SvPOK (SV* SV)
```

SvPOK_off
Unsets the PV status of an SV.

```
void    SvPOK_off (SV* sv)
```

SvPOK_on
Tells an SV that it is a string.

```
void    SvPOK_on (SV* sv)
```

SvPOK_only
Tells an SV that it is a string and disables all other OK bits.

```
void    SvPOK_only (SV* sv)
```

SvPOKp
Returns a boolean indicating whether the SV contains a character string. Checks the **private** setting. Use `SvPOK`.

```
int     SvPOKp (SV* SV)
```

SvPV Returns a pointer to the string in the SV, or a stringified form of the SV if the SV does not contain a string. If `len` is `PL_na` then Perl will handle the length on its own. Handles 'get' magic.

```
char*   SvPV (SV* sv, int len )
```

SvPV_force
Like <SvPV> but will force the SV into becoming a string (SvPOK). You want force if you are going to update the SvPVX directly.

```
char*   SvPV_force(SV* sv, int len)
```

SvPVX Returns a pointer to the string in the SV. The SV must contain a string.

```
char*   SvPVX (SV* sv)
```

SvREFCNT
Returns the value of the object's reference count.

```
int     SvREFCNT (SV* sv)
```

SvREFCNT_dec
Decrements the reference count of the given SV.

```
void    SvREFCNT_dec (SV* sv)
```

SvREFCNT_inc
Increments the reference count of the given SV.

```
void    SvREFCNT_inc (SV* sv)
```

SvROK Tests if the SV is an RV.

```
int     SvROK (SV* sv)
```

SvROK_off
Unsets the RV status of an SV.

```
void    SvROK_off (SV* sv)
```

SvROK_on
Tells an SV that it is an RV.

```
void    SvROK_on (SV* sv)
```

SvRV Dereferences an RV to return the SV.

```
SV*     SvRV (SV* sv)
```

SvSETMAGIC
Invokes `mg_set` on an SV if it has 'set' magic. This macro evaluates its argument more than once.

```
void    SvSETMAGIC( SV *sv )
```

sv_setiv Copies an integer into the given SV. Does not handle 'set' magic. See `sv_setiv_mg`.

```
void    sv_setiv (SV* sv, IV num)
```

sv_setiv_mg
Like `sv_setiv`, but also handles 'set' magic.

```
void    sv_setiv_mg (SV* sv, IV num)
```

sv_setnv Copies a double into the given SV. Does not handle 'set' magic. See `sv_setnv_mg`.

```
void    sv_setnv (SV* sv, double num)
```

sv_setnv_mg
Like `sv_setnv`, but also handles 'set' magic.

```
void    sv_setnv_mg (SV* sv, double num)
```

sv_setpv
Copies a string into an SV. The string must be null-terminated. Does not handle 'set' magic. See `sv_setpv_mg`.

```
void    sv_setpv (SV* sv, char* ptr)
```

sv_setpv_mg
Like `sv_setpv`, but also handles 'set' magic.

```
void    sv_setpv_mg (SV* sv, char* ptr)
```

sv_setpviv
Copies an integer into the given SV, also updating its string value. Does not handle 'set' magic. See `sv_setpviv_mg`.

```
void    sv_setpviv (SV* sv, IV num)
```

sv_setpviv_mg
Like `sv_setpviv`, but also handles 'set' magic.

```
void    sv_setpviv_mg (SV* sv, IV num)
```

sv_setpvn
Copies a string into an SV. The `len` parameter indicates the number of bytes to be copied. Does not handle 'set' magic. See `sv_setpvn_mg`.

```
void    sv_setpvn (SV* sv, char* ptr, STRLEN len)
```

sv_setpvn_mg
Like `sv_setpvn`, but also handles 'set' magic.

```
void    sv_setpvn_mg (SV* sv, char* ptr, STRLEN len)
```

sv_setpvf
Processes its arguments like `sprintf` and sets an SV to the formatted output. Does not handle 'set' magic. See `sv_setpvf_mg`.

```
void    sv_setpvf (SV* sv, const char* pat, ...)
```

sv_setpvf_mg
Like `sv_setpvf`, but also handles 'set' magic.

```
void    sv_setpvf_mg (SV* sv, const char* pat, ...)
```

sv_setref_iv
Copies an integer into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

```
SV*     sv_setref_iv (SV *rv, char *classname, IV iv)
```

sv_setref_nv
Copies a double into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

```
SV*     sv_setref_nv (SV *rv, char *classname, double nv)
```

sv_setref_pv
Copies a pointer into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. If the `pv` argument is NULL then `PL_sv_undef` will be placed into the SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

```
SV*     sv_setref_pv (SV *rv, char *classname, void* pv)
```

Do not use with integral Perl types such as HV, AV, SV, CV, because those objects will become corrupted by the pointer copy process.

Note that `sv_setref_pvn` copies the string while this copies the pointer.

sv_setref_pvn
Copies a string into a new SV, optionally blessing the SV. The length of the string must be specified with `n`. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

```
SV*     sv_setref_pvn (SV *rv, char *classname, char* pv, I32 n)
```

Note that `sv_setref_pv` copies the pointer while this copies the string.

SvSetSV
Calls `sv_setsv` if dsv is not the same as ssv. May evaluate arguments more than once.

```
void    SvSetSV (SV* dsv, SV* ssv)
```

SvSetSV_nosteal
Calls a non-destructive version of `sv_setsv` if dsv is not the same as ssv. May evaluate arguments more than once.

```
void    SvSetSV_nosteal (SV* dsv, SV* ssv)
```

sv_setsv
Copies the contents of the source SV `ssv` into the destination SV `dsv`. The source SV may be destroyed if it is mortal. Does not handle 'set' magic. See the macro forms `SvSetSV`, `SvSetSV_nosteal` and `sv_setsv_mg`.

```
void    sv_setsv (SV* dsv, SV* ssv)
```

sv_setsv_mg
Like `sv_setsv`, but also handles 'set' magic.

```
void    sv_setsv_mg (SV* dsv, SV* ssv)
```

sv_setuv Copies an unsigned integer into the given SV. Does not handle 'set' magic. See `sv_setuv_mg`.

```
void    sv_setuv (SV* sv, UV num)
```

sv_setuv_mg
Like `sv_setuv`, but also handles 'set' magic.

```
void    sv_setuv_mg (SV* sv, UV num)
```

SvSTASH
Returns the stash of the SV.

```
HV*     SvSTASH (SV* sv)
```

SvTAINT
Taints an SV if tainting is enabled

```
void    SvTAINT (SV* sv)
```

SvTAINTED
Checks to see if an SV is tainted. Returns TRUE if it is, FALSE if not.

```
int     SvTAINTED (SV* sv)
```

SvTAINTED_off
Untaints an SV. Be *very* careful with this routine, as it short-circuits some of Perl's fundamental security features. XS module authors should not use this function unless they fully understand all the implications of unconditionally untainting the value. Untainting should be done in the standard perl fashion, via a carefully crafted regexp, rather than directly untainting variables.

```
void    SvTAINTED_off (SV* sv)
```

SvTAINTED_on
Marks an SV as tainted.

```
void    SvTAINTED_on (SV* sv)
```

SVt_IV Integer type flag for scalars. See `svtype`.

SVt_PV Pointer type flag for scalars. See `svtype`.

SVt_PVAV
Type flag for arrays. See `svtype`.

SVt_PVCV
Type flag for code refs. See `svtype`.

SVt_PVHV
Type flag for hashes. See `svtype`.

SVt_PVMG
Type flag for blessed scalars. See `svtype`.

SVt_NV Double type flag for scalars. See `svtype`.

SvTRUE
Returns a boolean indicating whether Perl would evaluate the SV as true or false, defined or undefined. Does not handle 'get' magic.

```
int     SvTRUE (SV* sv)
```

SvTYPE
Returns the type of the SV. See `svtype`.

```
svtype  SvTYPE (SV* sv)
```

svtype An enum of flags for Perl types. These are found in the file **sv.h** in the `svtype` enum. Test these flags with the `SvTYPE` macro.

PL_sv_undef
This is the `undef` SV. Always refer to this as `&PL_sv_undef`.

sv_unref Unsets the RV status of the SV, and decrements the reference count of whatever was being referenced by the RV. This can almost be thought of as a reversal of `newSVrv`. See `SvROK_off`.

```
void    sv_unref (SV* sv)
```

SvUPGRADE
Used to upgrade an SV to a more complex form. Uses `sv_upgrade` to perform the upgrade if necessary. See `svtype`.

```
bool    SvUPGRADE (SV* sv, svtype mt)
```

sv_upgrade
Upgrade an SV to a more complex form. Use `SvUPGRADE`. See `svtype`.

sv_usepvn
Tells an SV to use `ptr` to find its string value. Normally the string is stored inside the SV but sv_usepvn allows the SV to use an outside string. The `ptr` should point to memory that was allocated by `malloc`. The string length, `len`, must be supplied. This function will realloc the memory pointed to by `ptr`, so that pointer should not be freed or used by the programmer after giving it to sv_usepvn. Does not handle 'set' magic. See `sv_usepvn_mg`.

```
void    sv_usepvn (SV* sv, char* ptr, STRLEN len)
```

sv_usepvn_mg
Like `sv_usepvn`, but also handles 'set' magic.

```
void    sv_usepvn_mg (SV* sv, char* ptr, STRLEN len)
```

sv_vcatpvfn(sv, pat, patlen, args, svargs, svmax, used_locale)
Processes its arguments like `vsprintf` and appends the formatted output to an SV. Uses an array of SVs if the C style variable argument list is missing (NULL). Indicates if locale information has been used for formatting.

```
void    sv_catpvfn _((SV* sv, const char* pat, STRLEN patlen,
                      va_list *args, SV **svargs, I32 svmax,
                      bool *used_locale));
```

sv_vsetpvfn(sv, pat, patlen, args, svargs, svmax, used_locale)
Works like `vcatpvfn` but copies the text into the SV instead of appending it.

```
void    sv_setpvfn _((SV* sv, const char* pat, STRLEN patlen,
                      va_list *args, SV **svargs, I32 svmax,
                      bool *used_locale));
```

SvUV Returns the unsigned integer which is in the SV.

```
UV      SvUV(SV* sv)
```

SvUVX Returns the unsigned integer which is stored in the SV.

```
UV      SvUVX(SV* sv)
```

PL_sv_yes
This is the `true` SV. See `PL_sv_no`. Always refer to this as `&PL_sv_yes`.

THIS Variable which is setup by `xsubpp` to designate the object in a C++ XSUB. This is always the proper type for the C++ object. See `CLASS` and the section on *Using XS With C++* in the *perlxs* manpage.

toLOWER
Converts the specified character to lowercase.

```
int     toLOWER (char c)
```

toUPPER
Converts the specified character to uppercase.

```
int     toUPPER (char c)
```

warn This is the XSUB−writer's interface to Perl's `warn` function. Use this function the same way you use the C `printf` function. See `croak()`.

XPUSHi
Push an integer onto the stack, extending the stack if necessary. Handles 'set' magic. See `PUSHi`.

```
XPUSHi(int d)
```

XPUSHn
Push a double onto the stack, extending the stack if necessary. Handles 'set' magic. See `PUSHn`.

```
XPUSHn(double d)
```

XPUSHp
Push a string onto the stack, extending the stack if necessary. The `len` indicates the length of the string. Handles 'set' magic. See `PUSHp`.

```
XPUSHp(char *c, int len)
```

XPUSHs
Push an SV onto the stack, extending the stack if necessary. Does not handle 'set' magic. See `PUSHs`.

```
XPUSHs(sv)
```

XPUSHu
Push an unsigned integer onto the stack, extending the stack if necessary. See `PUSHu`.

XS Macro to declare an XSUB and its C parameter list. This is handled by `xsubpp`.

XSRETURN
: Return from XSUB, indicating number of items on the stack. This is usually handled by `xsubpp`.

```
XSRETURN(int x)
```

XSRETURN_EMPTY
: Return an empty list from an XSUB immediately.

```
XSRETURN_EMPTY;
```

XSRETURN_IV
: Return an integer from an XSUB immediately. Uses `XST_mIV`.

```
XSRETURN_IV(IV v)
```

XSRETURN_NO
: Return `&PL_sv_no` from an XSUB immediately. Uses `XST_mNO`.

```
XSRETURN_NO;
```

XSRETURN_NV
: Return an double from an XSUB immediately. Uses `XST_mNV`.

```
XSRETURN_NV(NV v)
```

XSRETURN_PV
: Return a copy of a string from an XSUB immediately. Uses `XST_mPV`.

```
XSRETURN_PV(char *v)
```

XSRETURN_UNDEF
: Return `&PL_sv_undef` from an XSUB immediately. Uses `XST_mUNDEF`.

```
XSRETURN_UNDEF;
```

XSRETURN_YES
: Return `&PL_sv_yes` from an XSUB immediately. Uses `XST_mYES`.

```
XSRETURN_YES;
```

XST_mIV
: Place an integer into the specified position `i` on the stack. The value is stored in a new mortal SV.

```
XST_mIV( int i, IV v )
```

XST_mNV
: Place a double into the specified position `i` on the stack. The value is stored in a new mortal SV.

```
XST_mNV( int i, NV v )
```

XST_mNO
: Place `&PL_sv_no` into the specified position `i` on the stack.

```
XST_mNO( int i )
```

XST_mPV
Place a copy of a string into the specified position `i` on the stack. The value is stored in a new mortal SV.

```
XST_mPV( int i, char *v )
```

XST_mUNDEF
Place `&PL_sv_undef` into the specified position `i` on the stack.

```
XST_mUNDEF( int i )
```

XST_mYES
Place `&PL_sv_yes` into the specified position `i` on the stack.

```
XST_mYES( int i )
```

XS_VERSION
The version identifier for an XS module. This is usually handled automatically by `ExtUtils::MakeMaker`. See `XS_VERSION_BOOTCHECK`.

XS_VERSION_BOOTCHECK
Macro to verify that a PM module's `$VERSION` variable matches the XS module's `XS_VERSION` variable. This is usually handled automatically by `xsubpp`. See the section on *The VERSIONCHECK: Keyword* in the *perlxs* manpage.

Zero
The XSUB−writer's interface to the C `memzero` function. The `d` is the destination, `n` is the number of items, and `t` is the type.

```
void    Zero( d, n, t )
```

**AUTHORS**

Until May 1997, this document was maintained by Jeff Okamoto <okamoto@corp.hp.com>. It is now maintained as part of Perl itself.

With lots of help and suggestions from Dean Roehrich, Malcolm Beattie, Andreas Koenig, Paul Hudson, Ilya Zakharevich, Paul Marquess, Neil Bowers, Matthew Green, Tim Bunce, Spider Boardman, Ulrich Pfeifer, Stephen McCamant, and Gurusamy Sarathy.

API Listing originally by Dean Roehrich <roehrich@cray.com>.

## NAME

perlcall – Perl calling conventions from C

## DESCRIPTION

The purpose of this document is to show you how to call Perl subroutines directly from C, i.e., how to write *callbacks*.

Apart from discussing the C interface provided by Perl for writing callbacks the document uses a series of examples to show how the interface actually works in practice. In addition some techniques for coding callbacks are covered.

Examples where callbacks are necessary include

- An Error Handler

  You have created an XSUB interface to an application's C API.

  A fairly common feature in applications is to allow you to define a C function that will be called whenever something nasty occurs. What we would like is to be able to specify a Perl subroutine that will be called instead.

- An Event Driven Program

  The classic example of where callbacks are used is when writing an event driven program like for an X windows application. In this case you register functions to be called whenever specific events occur, e.g., a mouse button is pressed, the cursor moves into a window or a menu item is selected.

Although the techniques described here are applicable when embedding Perl in a C program, this is not the primary goal of this document. There are other details that must be considered and are specific to embedding Perl. For details on embedding Perl in C refer to the *perlembed* manpage.

Before you launch yourself head first into the rest of this document, it would be a good idea to have read the following two documents – the *perlxs* manpage and the *perlguts* manpage.

## THE PERL_CALL FUNCTIONS

Although this stuff is easier to explain using examples, you first need be aware of a few important definitions.

Perl has a number of C functions that allow you to call Perl subroutines. They are

```
I32 perl_call_sv(SV* sv, I32 flags) ;
I32 perl_call_pv(char *subname, I32 flags) ;
I32 perl_call_method(char *methname, I32 flags) ;
I32 perl_call_argv(char *subname, I32 flags, register char **argv) ;
```

The key function is *perl_call_sv*. All the other functions are fairly simple wrappers which make it easier to call Perl subroutines in special cases. At the end of the day they will all call *perl_call_sv* to invoke the Perl subroutine.

All the *perl_call_** functions have a `flags` parameter which is used to pass a bit mask of options to Perl. This bit mask operates identically for each of the functions. The settings available in the bit mask are discussed in the section on *FLAG VALUES*.

Each of the functions will now be discussed in turn.

### perl_call_sv

*perl_call_sv* takes two parameters, the first, `sv`, is an SV*. This allows you to specify the Perl subroutine to be called either as a C string (which has first been converted to an SV) or a reference to a subroutine. The section, *Using perl_call_sv*, shows how you can make use of *perl_call_sv*.

### perl_call_pv

The function, *perl_call_pv*, is similar to *perl_call_sv* except it expects its first parameter to be a C char* which identifies the Perl subroutine you want to call, e.g., `perl_call_pv("fred", 0)`. If the subroutine you want to call is in another package, just include the package name in the string, e.g., `"pkg::fred"`.

**perl_call_method**

The function *perl_call_method* is used to call a method from a Perl class. The parameter `methname` corresponds to the name of the method to be called. Note that the class that the method belongs to is passed on the Perl stack rather than in the parameter list. This class can be either the name of the class (for a static method) or a reference to an object (for a virtual method). See the *perlobj* manpage for more information on static and virtual methods and the section on *Using perl_call_method* for an example of using *perl_call_method*.

**perl_call_argv**

*perl_call_argv* calls the Perl subroutine specified by the C string stored in the `subname` parameter. It also takes the usual `flags` parameter. The final parameter, `argv`, consists of a NULL terminated list of C strings to be passed as parameters to the Perl subroutine. See *Using perl_call_argv*.

All the functions return an integer. This is a count of the number of items returned by the Perl subroutine. The actual items returned by the subroutine are stored on the Perl stack.

As a general rule you should *always* check the return value from these functions. Even if you are expecting only a particular number of values to be returned from the Perl subroutine, there is nothing to stop someone from doing something unexpected – don't say you haven't been warned.

## FLAG VALUES

The `flags` parameter in all the *perl_call_** functions is a bit mask which can consist of any combination of the symbols defined below, OR'ed together.

### G_VOID

Calls the Perl subroutine in a void context.

This flag has 2 effects:

1. It indicates to the subroutine being called that it is executing in a void context (if it executes *wantarray* the result will be the undefined value).
2. It ensures that nothing is actually returned from the subroutine.

The value returned by the *perl_call_** function indicates how many items have been returned by the Perl subroutine – in this case it will be 0.

### G_SCALAR

Calls the Perl subroutine in a scalar context. This is the default context flag setting for all the *perl_call_** functions.

This flag has 2 effects:

1. It indicates to the subroutine being called that it is executing in a scalar context (if it executes *wantarray* the result will be false).
2. It ensures that only a scalar is actually returned from the subroutine. The subroutine can, of course, ignore the *wantarray* and return a list anyway. If so, then only the last element of the list will be returned.

The value returned by the *perl_call_** function indicates how many items have been returned by the Perl subroutine – in this case it will be either 0 or 1.

If 0, then you have specified the G_DISCARD flag.

If 1, then the item actually returned by the Perl subroutine will be stored on the Perl stack – the section *Returning a Scalar* shows how to access this value on the stack. Remember that regardless of how many items the Perl subroutine returns, only the last one will be accessible from the stack – think of the case where only one value is returned as being a list with only one element. Any other items that were returned will not exist by the time control returns from the *perl_call_** function. The section *Returning a list in a scalar context* shows an example of this behavior.

**G_ARRAY**

Calls the Perl subroutine in a list context.

As with G_SCALAR, this flag has 2 effects:

1. It indicates to the subroutine being called that it is executing in an array context (if it executes *wantarray* the result will be true).
2. It ensures that all items returned from the subroutine will be accessible when control returns from the *perl_call_** function.

The value returned by the *perl_call_** function indicates how many items have been returned by the Perl subroutine.

If 0, then you have specified the G_DISCARD flag.

If not 0, then it will be a count of the number of items returned by the subroutine. These items will be stored on the Perl stack. The section *Returning a list of values* gives an example of using the G_ARRAY flag and the mechanics of accessing the returned items from the Perl stack.

**G_DISCARD**

By default, the *perl_call_** functions place the items returned from by the Perl subroutine on the stack. If you are not interested in these items, then setting this flag will make Perl get rid of them automatically for you. Note that it is still possible to indicate a context to the Perl subroutine by using either G_SCALAR or G_ARRAY.

If you do not set this flag then it is *very* important that you make sure that any temporaries (i.e., parameters passed to the Perl subroutine and values returned from the subroutine) are disposed of yourself. The section *Returning a Scalar* gives details of how to dispose of these temporaries explicitly and the section *Using Perl to dispose of temporaries* discusses the specific circumstances where you can ignore the problem and let Perl deal with it for you.

**G_NOARGS**

Whenever a Perl subroutine is called using one of the *perl_call_** functions, it is assumed by default that parameters are to be passed to the subroutine. If you are not passing any parameters to the Perl subroutine, you can save a bit of time by setting this flag. It has the effect of not creating the `@_` array for the Perl subroutine.

Although the functionality provided by this flag may seem straightforward, it should be used only if there is a good reason to do so. The reason for being cautious is that even if you have specified the G_NOARGS flag, it is still possible for the Perl subroutine that has been called to think that you have passed it parameters.

In fact, what can happen is that the Perl subroutine you have called can access the `@_` array from a previous Perl subroutine. This will occur when the code that is executing the *perl_call_** function has itself been called from another Perl subroutine. The code below illustrates this

```
sub fred
  { print "@_\n"  }

sub joe
  { &fred }

&joe(1,2,3) ;
```

This will print

```
1 2 3
```

What has happened is that `fred` accesses the `@_` array which belongs to `joe`.

### G_EVAL

It is possible for the Perl subroutine you are calling to terminate abnormally, e.g., by calling *die* explicitly or by not actually existing. By default, when either of these events occurs, the process will terminate immediately. If you want to trap this type of event, specify the G_EVAL flag. It will put an *eval { }* around the subroutine call.

Whenever control returns from the *perl_call_** function you need to check the `$@` variable as you would in a normal Perl script.

The value returned from the *perl_call_** function is dependent on what other flags have been specified and whether an error has occurred. Here are all the different cases that can occur:

- If the *perl_call_** function returns normally, then the value returned is as specified in the previous sections.
- If G_DISCARD is specified, the return value will always be 0.
- If G_ARRAY is specified *and* an error has occurred, the return value will always be 0.
- If G_SCALAR is specified *and* an error has occurred, the return value will be 1 and the value on the top of the stack will be *undef*. This means that if you have already detected the error by checking `$@` and you want the program to continue, you must remember to pop the *undef* from the stack.

See *Using G_EVAL* for details on using G_EVAL.

### G_KEEPERR

You may have noticed that using the G_EVAL flag described above will **always** clear the `$@` variable and set it to a string describing the error iff there was an error in the called code. This unqualified resetting of `$@` can be problematic in the reliable identification of errors using the `eval {}` mechanism, because the possibility exists that perl will call other code (end of block processing code, for example) between the time the error causes `$@` to be set within `eval {}`, and the subsequent statement which checks for the value of `$@` gets executed in the user's script.

This scenario will mostly be applicable to code that is meant to be called from within destructors, asynchronous callbacks, signal handlers, `__DIE__` or `__WARN__` hooks, and `tie` functions. In such situations, you will not want to clear `$@` at all, but simply to append any new errors to any existing value of `$@`.

The G_KEEPERR flag is meant to be used in conjunction with G_EVAL in *perl_call_** functions that are used to implement such code. This flag has no effect when G_EVAL is not used.

When G_KEEPERR is used, any errors in the called code will be prefixed with the string "\t (in cleanup)", and appended to the current value of `$@`.

The G_KEEPERR flag was introduced in Perl version 5.002.

See *Using G_KEEPERR* for an example of a situation that warrants the use of this flag.

### Determining the Context

As mentioned above, you can determine the context of the currently executing subroutine in Perl with *wantarray*. The equivalent test can be made in C by using the `GIMME_V` macro, which returns `G_ARRAY` if you have been called in an array context, `G_SCALAR` if in a scalar context, or `G_VOID` if in a void context (i.e. the return value will not be used). An older version of this macro is called `GIMME`; in a void context it returns `G_SCALAR` instead of `G_VOID`. An example of using the `GIMME_V` macro is shown in section *Using GIMME_V*.

## KNOWN PROBLEMS

This section outlines all known problems that exist in the *perl_call_** functions.

1. If you are intending to make use of both the G_EVAL and G_SCALAR flags in your code, use a version of Perl greater than 5.000. There is a bug in version 5.000 of Perl which means that the combination of these two flags will not work as described in the section *FLAG VALUES*.

Specifically, if the two flags are used when calling a subroutine and that subroutine does not call *die*, the value returned by *perl_call_** will be wrong.

2. In Perl 5.000 and 5.001 there is a problem with using *perl_call_** if the Perl sub you are calling attempts to trap a *die*.

   The symptom of this problem is that the called Perl sub will continue to completion, but whenever it attempts to pass control back to the XSUB, the program will immediately terminate.

   For example, say you want to call this Perl sub

```
sub fred
{
    eval { die "Fatal Error" ; }
    print "Trapped error: $@\n"
        if $@ ;
}
```

   via this XSUB

```
void
Call_fred()
    CODE:
    PUSHMARK(SP) ;
    perl_call_pv("fred", G_DISCARD|G_NOARGS) ;
    fprintf(stderr, "back in Call_fred\n") ;
```

   When `Call_fred` is executed it will print

```
Trapped error: Fatal Error
```

   As control never returns to `Call_fred`, the `"back in Call_fred"` string will not get printed.

   To work around this problem, you can either upgrade to Perl 5.002 or higher, or use the G_EVAL flag with *perl_call_** as shown below

```
void
Call_fred()
    CODE:
    PUSHMARK(SP) ;
    perl_call_pv("fred", G_EVAL|G_DISCARD|G_NOARGS) ;
    fprintf(stderr, "back in Call_fred\n") ;
```

## EXAMPLES

Enough of the definition talk, let's have a few examples.

Perl provides many macros to assist in accessing the Perl stack. Wherever possible, these macros should always be used when interfacing to Perl internals. We hope this should make the code less vulnerable to any changes made to Perl in the future.

Another point worth noting is that in the first series of examples I have made use of only the *perl_call_pv* function. This has been done to keep the code simpler and ease you into the topic. Wherever possible, if the choice is between using *perl_call_pv* and *perl_call_sv*, you should always try to use *perl_call_sv*. See *Using perl_call_sv* for details.

### No Parameters, Nothing returned

This first trivial example will call a Perl subroutine, *PrintUID*, to print out the UID of the process.

```
sub PrintUID
{
    print "UID is $<\n" ;
}
```

and here is a C function to call it

```
static void
call_PrintUID()
{
    dSP ;

    PUSHMARK(SP) ;
    perl_call_pv("PrintUID", G_DISCARD|G_NOARGS) ;
}
```

Simple, eh.

A few points to note about this example.

1. Ignore `dSP` and `PUSHMARK(SP)` for now. They will be discussed in the next example.
2. We aren't passing any parameters to *PrintUID* so G_NOARGS can be specified.
3. We aren't interested in anything returned from *PrintUID*, so G_DISCARD is specified. Even if *PrintUID* was changed to return some *value* (s), having specified G_DISCARD will mean that they will be wiped by the time control returns from *perl_call_pv*.
4. As *perl_call_pv* is being used, the Perl subroutine is specified as a C string. In this case the subroutine name has been 'hard-wired' into the code.
5. Because we specified G_DISCARD, it is not necessary to check the value returned from *perl_call_pv*. It will always be 0.

### Passing Parameters

Now let's make a slightly more complex example. This time we want to call a Perl subroutine, `LeftString`, which will take 2 parameters – a string (`$s`) and an integer (`$n`). The subroutine will simply print the first `$n` characters of the string.

So the Perl subroutine would look like this

```
sub LeftString
{
    my($s, $n) = @_ ;
    print substr($s, 0, $n), "\n" ;
}
```

The C function required to call *LeftString* would look like this.

```
static void
call_LeftString(a, b)
char * a ;
int b ;
{
    dSP ;
```

```
        ENTER ;
        SAVETMPS ;

        PUSHMARK(SP) ;
        XPUSHs(sv_2mortal(newSVpv(a, 0)));
        XPUSHs(sv_2mortal(newSViv(b)));
        PUTBACK ;

        perl_call_pv("LeftString", G_DISCARD);

        FREETMPS ;
        LEAVE ;
    }
```

Here are a few notes on the C function *call_LeftString*.

1. Parameters are passed to the Perl subroutine using the Perl stack. This is the purpose of the code beginning with the line `dSP` and ending with the line `PUTBACK`. The `dSP` declares a local copy of the stack pointer. This local copy should **always** be accessed as `SP`.

2. If you are going to put something onto the Perl stack, you need to know where to put it. This is the purpose of the macro `dSP` – it declares and initializes a *local* copy of the Perl stack pointer.

   All the other macros which will be used in this example require you to have used this macro.

   The exception to this rule is if you are calling a Perl subroutine directly from an XSUB function. In this case it is not necessary to use the `dSP` macro explicitly – it will be declared for you automatically.

3. Any parameters to be pushed onto the stack should be bracketed by the `PUSHMARK` and `PUTBACK` macros. The purpose of these two macros, in this context, is to count the number of parameters you are pushing automatically. Then whenever Perl is creating the `@_` array for the subroutine, it knows how big to make it.

   The `PUSHMARK` macro tells Perl to make a mental note of the current stack pointer. Even if you aren't passing any parameters (like the example shown in the section *No Parameters, Nothing returned*) you must still call the `PUSHMARK` macro before you can call any of the *perl_call_** functions – Perl still needs to know that there are no parameters.

   The `PUTBACK` macro sets the global copy of the stack pointer to be the same as our local copy. If we didn't do this *perl_call_pv* wouldn't know where the two parameters we pushed were – remember that up to now all the stack pointer manipulation we have done is with our local copy, *not* the global copy.

4. The only flag specified this time is G_DISCARD. Because we are passing 2 parameters to the Perl subroutine this time, we have not specified G_NOARGS.

5. Next, we come to XPUSHs. This is where the parameters actually get pushed onto the stack. In this case we are pushing a string and an integer.

   See the section on *XSUBs and the Argument Stack* in the *perlguts* manpage for details on how the XPUSH macros work.

6. Because we created temporary values (by means of *sv_2mortal()* calls) we will have to tidy up the Perl stack and dispose of mortal SVs.

   This is the purpose of

   ```
   ENTER ;
   SAVETMPS ;
   ```

   at the start of the function, and

```
FREETMPS ;
LEAVE ;
```

at the end. The ENTER/SAVETMPS pair creates a boundary for any temporaries we create. This means that the temporaries we get rid of will be limited to those which were created after these calls.

The FREETMPS/LEAVE pair will get rid of any values returned by the Perl subroutine (see next example), plus it will also dump the mortal SVs we have created. Having ENTER/SAVETMPS at the beginning of the code makes sure that no other mortals are destroyed.

Think of these macros as working a bit like using { and } in Perl to limit the scope of local variables.

See the section *Using Perl to dispose of temporaries* for details of an alternative to using these macros.

7. Finally, *LeftString* can now be called via the *perl_call_pv* function.

**Returning a Scalar**

Now for an example of dealing with the items returned from a Perl subroutine.

Here is a Perl subroutine, *Adder*, that takes 2 integer parameters and simply returns their sum.

```
sub Adder
{
    my($a, $b) = @_ ;
    $a + $b ;
}
```

Because we are now concerned with the return value from *Adder*, the C function required to call it is now a bit more complex.

```
static void
call_Adder(a, b)
int a ;
int b ;
{
    dSP ;
    int count ;

    ENTER ;
    SAVETMPS;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK ;

    count = perl_call_pv("Adder", G_SCALAR);

    SPAGAIN ;

    if (count != 1)
        croak("Big trouble\n") ;

    printf ("The sum of %d and %d is %d\n", a, b, POPi) ;
```

```
        PUTBACK ;
        FREETMPS ;
        LEAVE ;
    }
```

Points to note this time are

1. The only flag specified this time was G_SCALAR. That means the `@_` array will be created and that the value returned by *Adder* will still exist after the call to *perl_call_pv*.

2. The purpose of the macro `SPAGAIN` is to refresh the local copy of the stack pointer. This is necessary because it is possible that the memory allocated to the Perl stack has been reallocated whilst in the *perl_call_pv* call.

   If you are making use of the Perl stack pointer in your code you must always refresh the local copy using SPAGAIN whenever you make use of the *perl_call_** functions or any other Perl internal function.

3. Although only a single value was expected to be returned from *Adder*, it is still good practice to check the return code from *perl_call_pv* anyway.

   Expecting a single value is not quite the same as knowing that there will be one. If someone modified *Adder* to return a list and we didn't check for that possibility and take appropriate action the Perl stack would end up in an inconsistent state. That is something you *really* don't want to happen ever.

4. The `POPi` macro is used here to pop the return value from the stack. In this case we wanted an integer, so `POPi` was used.

   Here is the complete list of POP macros available, along with the types they return.

```
    POPs        SV
    POPp        pointer
    POPn        double
    POPi        integer
    POPl        long
```

5. The final `PUTBACK` is used to leave the Perl stack in a consistent state before exiting the function. This is necessary because when we popped the return value from the stack with `POPi` it updated only our local copy of the stack pointer. Remember, `PUTBACK` sets the global stack pointer to be the same as our local copy.

### Returning a list of values

Now, let's extend the previous example to return both the sum of the parameters and the difference.

Here is the Perl subroutine

```
    sub AddSubtract
    {
       my($a, $b) = @_ ;
       ($a+$b, $a-$b) ;
    }
```

and this is the C function

```
static void
call_AddSubtract(a, b)
int a ;
int b ;
{
    dSP ;
    int count ;

    ENTER ;
    SAVETMPS;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK ;

    count = perl_call_pv("AddSubtract", G_ARRAY);

    SPAGAIN ;

    if (count != 2)
        croak("Big trouble\n") ;

    printf ("%d - %d = %d\n", a, b, POPi) ;
    printf ("%d + %d = %d\n", a, b, POPi) ;

    PUTBACK ;
    FREETMPS ;
    LEAVE ;
}
```

If *call_AddSubtract* is called like this

```
call_AddSubtract(7, 4) ;
```

then here is the output

```
7 - 4 = 3
7 + 4 = 11
```

Notes

1. We wanted array context, so G_ARRAY was used.
2. Not surprisingly `POPi` is used twice this time because we were retrieving 2 values from the stack. The important thing to note is that when using the `POP*` macros they come off the stack in *reverse* order.

**Returning a list in a scalar context**

Say the Perl subroutine in the previous section was called in a scalar context, like this

```
static void
call_AddSubScalar(a, b)
int a ;
int b ;
{
    dSP ;
    int count ;
    int i ;

    ENTER ;
    SAVETMPS;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK ;

    count = perl_call_pv("AddSubtract", G_SCALAR);

    SPAGAIN ;

    printf ("Items Returned = %d\n", count) ;

    for (i = 1 ; i <= count ; ++i)
        printf ("Value %d = %d\n", i, POPi) ;

    PUTBACK ;
    FREETMPS ;
    LEAVE ;
}
```

The other modification made is that *call_AddSubScalar* will print the number of items returned from the Perl subroutine and their value (for simplicity it assumes that they are integer). So if *call_AddSubScalar* is called

```
call_AddSubScalar(7, 4) ;
```

then the output will be

```
Items Returned = 1
Value 1 = 3
```

In this case the main point to note is that only the last item in the list is returned from the subroutine, *AddSubtract* actually made it back to *call_AddSubScalar*.

### Returning Data from Perl via the parameter list

It is also possible to return values directly via the parameter list – whether it is actually desirable to do it is another matter entirely.

The Perl subroutine, *Inc*, below takes 2 parameters and increments each directly.

```
sub Inc
{
    ++ $_[0] ;
    ++ $_[1] ;
}
```

and here is a C function to call it.

```
static void
call_Inc(a, b)
int a ;
int b ;
{
    dSP ;
    int count ;
    SV * sva ;
    SV * svb ;

    ENTER ;
    SAVETMPS;

    sva = sv_2mortal(newSViv(a)) ;
    svb = sv_2mortal(newSViv(b)) ;

    PUSHMARK(SP) ;
    XPUSHs(sva);
    XPUSHs(svb);
    PUTBACK ;

    count = perl_call_pv("Inc", G_DISCARD);

    if (count != 0)
        croak ("call_Inc: expected 0 values from 'Inc', got %d\n",
               count) ;

    printf ("%d + 1 = %d\n", a, SvIV(sva)) ;
    printf ("%d + 1 = %d\n", b, SvIV(svb)) ;

    FREETMPS ;
    LEAVE ;
}
```

To be able to access the two parameters that were pushed onto the stack after they return from *perl_call_pv* it is necessary to make a note of their addresses – thus the two variables `sva` and `svb`.

The reason this is necessary is that the area of the Perl stack which held them will very likely have been overwritten by something else by the time control returns from *perl_call_pv*.

### Using G_EVAL

Now an example using G_EVAL. Below is a Perl subroutine which computes the difference of its 2 parameters. If this would result in a negative result, the subroutine calls *die*.

```
sub Subtract
{
    my ($a, $b) = @_ ;

    die "death can be fatal\n" if $a < $b ;

    $a - $b ;
}
```

and some C to call it

```
static void
call_Subtract(a, b)
int a ;
int b ;
{
    dSP ;
    int count ;

    ENTER ;
    SAVETMPS;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK ;

    count = perl_call_pv("Subtract", G_EVAL|G_SCALAR);

    SPAGAIN ;

    /* Check the eval first */
    if (SvTRUE(ERRSV))
    {
        printf ("Uh oh - %s\n", SvPV(ERRSV, PL_na)) ;
        POPs ;
    }
    else
    {
        if (count != 1)
           croak("call_Subtract: wanted 1 value from 'Subtract', got %d\n",
                    count) ;

        printf ("%d - %d = %d\n", a, b, POPi) ;
    }

    PUTBACK ;
    FREETMPS ;
    LEAVE ;
}
```

If *call_Subtract* is called thus

```
call_Subtract(4, 5)
```

the following will be printed

```
Uh oh - death can be fatal
```

Notes

1. We want to be able to catch the *die* so we have used the G_EVAL flag. Not specifying this flag would mean that the program would terminate immediately at the *die* statement in the subroutine *Subtract*.

2. The code

```
if (SvTRUE(ERRSV))
{
    printf ("Uh oh - %s\n", SvPV(ERRSV, PL_na)) ;
    POPs ;
}
```

is the direct equivalent of this bit of Perl

```
print "Uh oh - $@\n" if $@ ;
```

`PL_errgv` is a perl global of type `GV *` that points to the symbol table entry containing the error. `ERRSV` therefore refers to the C equivalent of `$@`.

3. Note that the stack is popped using `POPs` in the block where `SvTRUE(ERRSV)` is true. This is necessary because whenever a *perl_call_** function invoked with G_EVAL|G_SCALAR returns an error, the top of the stack holds the value *undef*. Because we want the program to continue after detecting this error, it is essential that the stack is tidied up by removing the *undef*.

**Using G_KEEPERR**

Consider this rather facetious example, where we have used an XS version of the call_Subtract example above inside a destructor:

```
package Foo;
sub new { bless {}, $_[0] }
sub Subtract {
    my($a,$b) = @_;
    die "death can be fatal" if $a < $b ;
    $a - $b;
}
sub DESTROY { call_Subtract(5, 4); }
sub foo { die "foo dies"; }

package main;
eval { Foo->new->foo };
print "Saw: $@" if $@;             # should be, but isn't
```

This example will fail to recognize that an error occurred inside the `eval {}`. Here's why: the call_Subtract code got executed while perl was cleaning up temporaries when exiting the eval block, and because call_Subtract is implemented with *perl_call_pv* using the G_EVAL flag, it promptly reset `$@`. This results in the failure of the outermost test for `$@`, and thereby the failure of the error trap.

Appending the G_KEEPERR flag, so that the *perl_call_pv* call in call_Subtract reads:

```
count = perl_call_pv("Subtract", G_EVAL|G_SCALAR|G_KEEPERR);
```

will preserve the error and restore reliable error handling.

**Using perl_call_sv**

In all the previous examples I have 'hard-wired' the name of the Perl subroutine to be called from C. Most of the time though, it is more convenient to be able to specify the name of the Perl subroutine from within the Perl script.

Consider the Perl code below

```
sub fred
{
    print "Hello there\n" ;
}

CallSubPV("fred") ;
```

Here is a snippet of XSUB which defines *CallSubPV*.

```
void
CallSubPV(name)
    char *  name
    CODE:
    PUSHMARK(SP) ;
    perl_call_pv(name, G_DISCARD|G_NOARGS) ;
```

That is fine as far as it goes. The thing is, the Perl subroutine can be specified as only a string. For Perl 4 this was adequate, but Perl 5 allows references to subroutines and anonymous subroutines. This is where *perl_call_sv* is useful.

The code below for *CallSubSV* is identical to *CallSubPV* except that the `name` parameter is now defined as an SV* and we use *perl_call_sv* instead of *perl_call_pv*.

```
void
CallSubSV(name)
    SV *    name
    CODE:
    PUSHMARK(SP) ;
    perl_call_sv(name, G_DISCARD|G_NOARGS) ;
```

Because we are using an SV to call *fred* the following can all be used

```
CallSubSV("fred") ;
CallSubSV(\&fred) ;
$ref = \&fred ;
CallSubSV($ref) ;
CallSubSV( sub { print "Hello there\n" } ) ;
```

As you can see, *perl_call_sv* gives you much greater flexibility in how you can specify the Perl subroutine.

You should note that if it is necessary to store the SV (`name` in the example above) which corresponds to the Perl subroutine so that it can be used later in the program, it not enough just to store a copy of the pointer to the SV. Say the code above had been like this

```
static SV * rememberSub ;

void
SaveSub1(name)
    SV *    name
    CODE:
    rememberSub = name ;

void
CallSavedSub1()
    CODE:
    PUSHMARK(SP) ;
    perl_call_sv(rememberSub, G_DISCARD|G_NOARGS) ;
```

The reason this is wrong is that by the time you come to use the pointer `rememberSub` in `CallSavedSub1`, it may or may not still refer to the Perl subroutine that was recorded in `SaveSub1`. This is particularly true for these cases

```
SaveSub1(\&fred) ;
CallSavedSub1() ;

SaveSub1( sub { print "Hello there\n" } ) ;
CallSavedSub1() ;
```

By the time each of the `SaveSub1` statements above have been executed, the SV*s which corresponded to the parameters will no longer exist. Expect an error message from Perl of the form

```
Can't use an undefined value as a subroutine reference at ...
```

for each of the `CallSavedSub1` lines.

Similarly, with this code

```
$ref = \&fred ;
SaveSub1($ref) ;
$ref = 47 ;
CallSavedSub1() ;
```

you can expect one of these messages (which you actually get is dependent on the version of Perl you are using)

```
Not a CODE reference at ...
Undefined subroutine &main::47 called ...
```

The variable `$ref` may have referred to the subroutine `fred` whenever the call to `SaveSub1` was made but by the time `CallSavedSub1` gets called it now holds the number `47`. Because we saved only a pointer to the original SV in `SaveSub1`, any changes to `$ref` will be tracked by the pointer `rememberSub`. This means that whenever `CallSavedSub1` gets called, it will attempt to execute the code which is referenced by the SV* `rememberSub`. In this case though, it now refers to the integer `47`, so expect Perl to complain loudly.

A similar but more subtle problem is illustrated with this code

```
$ref = \&fred ;
SaveSub1($ref) ;
$ref = \&joe ;
CallSavedSub1() ;
```

This time whenever `CallSavedSub1` get called it will execute the Perl subroutine `joe` (assuming it exists) rather than `fred` as was originally requested in the call to `SaveSub1`.

To get around these problems it is necessary to take a full copy of the SV. The code below shows `SaveSub2` modified to do that

```
static SV * keepSub = (SV*)NULL ;
```

```
void
SaveSub2(name)
    SV *    name
    CODE:
    /* Take a copy of the callback */
    if (keepSub == (SV*)NULL)
        /* First time, so create a new SV */
        keepSub = newSVsv(name) ;
    else
        /* Been here before, so overwrite */
        SvSetSV(keepSub, name) ;

void
CallSavedSub2()
    CODE:
    PUSHMARK(SP) ;
    perl_call_sv(keepSub, G_DISCARD|G_NOARGS) ;
```

To avoid creating a new SV every time `SaveSub2` is called, the function first checks to see if it has been called before. If not, then space for a new SV is allocated and the reference to the Perl subroutine, `name` is copied to the variable `keepSub` in one operation using `newSVsv`. Thereafter, whenever `SaveSub2` is called the existing SV, `keepSub`, is overwritten with the new value using `SvSetSV`.

**Using perl_call_argv**

Here is a Perl subroutine which prints whatever parameters are passed to it.

```
sub PrintList
{
    my(@list) = @_ ;

    foreach (@list) { print "$_\n" }
}
```

and here is an example of *perl_call_argv* which will call *PrintList*.

```
static char * words[] = {"alpha", "beta", "gamma", "delta", NULL} ;

static void
call_PrintList()
{
    dSP ;

    perl_call_argv("PrintList", G_DISCARD, words) ;
}
```

Note that it is not necessary to call `PUSHMARK` in this instance. This is because *perl_call_argv* will do it for you.

**Using perl_call_method**

Consider the following Perl code

```
{
    package Mine ;
```

```
        sub new
        {
            my($type) = shift ;
            bless [@_]
        }

        sub Display
        {
            my ($self, $index) = @_ ;
            print "$index: $$self[$index]\n" ;
        }

        sub PrintID
        {
            my($class) = @_ ;
            print "This is Class $class version 1.0\n" ;
        }
    }
```

It implements just a very simple class to manage an array. Apart from the constructor, `new`, it declares methods, one static and one virtual. The static method, `PrintID`, prints out simply the class name and a version number. The virtual method, `Display`, prints out a single element of the array. Here is an all Perl example of using it.

```
$a = new Mine ('red', 'green', 'blue') ;
$a->Display(1) ;
PrintID Mine;
```

will print

```
1: green
This is Class Mine version 1.0
```

Calling a Perl method from C is fairly straightforward. The following things are required

- a reference to the object for a virtual method or the name of the class for a static method.
- the name of the method.
- any other parameters specific to the method.

Here is a simple XSUB which illustrates the mechanics of calling both the `PrintID` and `Display` methods from C.

```
void
call_Method(ref, method, index)
    SV *    ref
    char *  method
    int             index
    CODE:
    PUSHMARK(SP);
    XPUSHs(ref);
    XPUSHs(sv_2mortal(newSViv(index))) ;
    PUTBACK;

    perl_call_method(method, G_DISCARD) ;
```

```
void
call_PrintID(class, method)
    char *  class
    char *  method
    CODE:
    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSVpv(class, 0))) ;
    PUTBACK;

    perl_call_method(method, G_DISCARD) ;
```

So the methods `PrintID` and `Display` can be invoked like this

```
$a = new Mine ('red', 'green', 'blue') ;
call_Method($a, 'Display', 1) ;
call_PrintID('Mine', 'PrintID') ;
```

The only thing to note is that in both the static and virtual methods, the method name is not passed via the stack – it is used as the first parameter to *perl_call_method*.

**Using GIMME_V**

Here is a trivial XSUB which prints the context in which it is currently executing.

```
void
PrintContext()
    CODE:
    I32 gimme = GIMME_V;
    if (gimme == G_VOID)
        printf ("Context is Void\n") ;
    else if (gimme == G_SCALAR)
        printf ("Context is Scalar\n") ;
    else
        printf ("Context is Array\n") ;
```

and here is some Perl to test it

```
PrintContext ;
$a = PrintContext ;
@a = PrintContext ;
```

The output from that will be

```
Context is Void
Context is Scalar
Context is Array
```

**Using Perl to dispose of temporaries**

In the examples given to date, any temporaries created in the callback (i.e., parameters passed on the stack to the *perl_call_** function or values returned via the stack) have been freed by one of these methods

- specifying the G_DISCARD flag with *perl_call_**.
- explicitly disposed of using the `ENTER`/`SAVETMPS` – `FREETMPS`/`LEAVE` pairing.

There is another method which can be used, namely letting Perl do it for you automatically whenever it regains control after the callback has terminated. This is done by simply not using the

```
ENTER ;
SAVETMPS ;
...
FREETMPS ;
LEAVE ;
```

sequence in the callback (and not, of course, specifying the G_DISCARD flag).

If you are going to use this method you have to be aware of a possible memory leak which can arise under very specific circumstances. To explain these circumstances you need to know a bit about the flow of control between Perl and the callback routine.

The examples given at the start of the document (an error handler and an event driven program) are typical of the two main sorts of flow control that you are likely to encounter with callbacks. There is a very important distinction between them, so pay attention.

In the first example, an error handler, the flow of control could be as follows. You have created an interface to an external library. Control can reach the external library like this

```
perl --> XSUB --> external library
```

Whilst control is in the library, an error condition occurs. You have previously set up a Perl callback to handle this situation, so it will get executed. Once the callback has finished, control will drop back to Perl again. Here is what the flow of control will be like in that situation

```
perl --> XSUB --> external library
                  ...
                  error occurs
                  ...
                  external library --> perl_call --> perl
                                                        |
perl <-- XSUB <-- external library <-- perl_call <----+
```

After processing of the error using *perl_call_** is completed, control reverts back to Perl more or less immediately.

In the diagram, the further right you go the more deeply nested the scope is. It is only when control is back with perl on the extreme left of the diagram that you will have dropped back to the enclosing scope and any temporaries you have left hanging around will be freed.

In the second example, an event driven program, the flow of control will be more like this

```
perl --> XSUB --> event handler
                  ...
                  event handler --> perl_call --> perl
                                                   |
                  event handler <-- perl_call <----+
                  ...
                  event handler --> perl_call --> perl
                                                   |
                  event handler <-- perl_call <----+
                  ...
                  event handler --> perl_call --> perl
                                                   |
                  event handler <-- perl_call <----+
```

In this case the flow of control can consist of only the repeated sequence

```
event handler --> perl_call --> perl
```

for practically the complete duration of the program. This means that control may *never* drop back to the surrounding scope in Perl at the extreme left.

So what is the big problem? Well, if you are expecting Perl to tidy up those temporaries for you, you might be in for a long wait. For Perl to dispose of your temporaries, control must drop back to the enclosing scope at some stage. In the event driven scenario that may never happen. This means that as time goes on, your program will create more and more temporaries, none of which will ever be freed. As each of these temporaries consumes some memory your program will eventually consume all the available memory in your system – kapow!

So here is the bottom line – if you are sure that control will revert back to the enclosing Perl scope fairly quickly after the end of your callback, then it isn't absolutely necessary to dispose explicitly of any temporaries you may have created. Mind you, if you are at all uncertain about what to do, it doesn't do any harm to tidy up anyway.

**Strategies for storing Callback Context Information**

Potentially one of the trickiest problems to overcome when designing a callback interface can be figuring out how to store the mapping between the C callback function and the Perl equivalent.

To help understand why this can be a real problem first consider how a callback is set up in an all C environment. Typically a C API will provide a function to register a callback. This will expect a pointer to a function as one of its parameters. Below is a call to a hypothetical function `register_fatal` which registers the C function to get called when a fatal error occurs.

```
register_fatal(cb1) ;
```

The single parameter `cb1` is a pointer to a function, so you must have defined `cb1` in your code, say something like this

```
static void
cb1()
{
    printf ("Fatal Error\n") ;
    exit(1) ;
}
```

Now change that to call a Perl subroutine instead

```
static SV * callback = (SV*)NULL;

static void
cb1()
{
    dSP ;

    PUSHMARK(SP) ;

    /* Call the Perl sub to process the callback */
    perl_call_sv(callback, G_DISCARD) ;
}
```

```
void
register_fatal(fn)
    SV *    fn
    CODE:
    /* Remember the Perl sub */
    if (callback == (SV*)NULL)
        callback = newSVsv(fn) ;
    else
        SvSetSV(callback, fn) ;

    /* register the callback with the external library */
    register_fatal(cb1) ;
```

where the Perl equivalent of `register_fatal` and the callback it registers, `pcb1`, might look like this

```
# Register the sub pcb1
register_fatal(\&pcb1) ;

sub pcb1
{
    die "I'm dying...\n" ;
}
```

The mapping between the C callback and the Perl equivalent is stored in the global variable `callback`.

This will be adequate if you ever need to have only one callback registered at any time. An example could be an error handler like the code sketched out above. Remember though, repeated calls to `register_fatal` will replace the previously registered callback function with the new one.

Say for example you want to interface to a library which allows asynchronous file i/o. In this case you may be able to register a callback whenever a read operation has completed. To be of any use we want to be able to call separate Perl subroutines for each file that is opened. As it stands, the error handler example above would not be adequate as it allows only a single callback to be defined at any time. What we require is a means of storing the mapping between the opened file and the Perl subroutine we want to be called for that file.

Say the i/o library has a function `asynch_read` which associates a C function `ProcessRead` with a file handle `fh` – this assumes that it has also provided some routine to open the file and so obtain the file handle.

```
asynch_read(fh, ProcessRead)
```

This may expect the C *ProcessRead* function of this form

```
void
ProcessRead(fh, buffer)
int fh ;
char *      buffer ;
{
     ...
}
```

To provide a Perl interface to this library we need to be able to map between the `fh` parameter and the Perl subroutine we want called. A hash is a convenient mechanism for storing this mapping. The code below shows a possible implementation

```
static HV * Mapping = (HV*)NULL ;
```

```
void
asynch_read(fh, callback)
    int     fh
    SV *    callback
    CODE:
    /* If the hash doesn't already exist, create it */
    if (Mapping == (HV*)NULL)
        Mapping = newHV() ;

    /* Save the fh -> callback mapping */
    hv_store(Mapping, (char*)&fh, sizeof(fh), newSVsv(callback), 0) ;

    /* Register with the C Library */
    asynch_read(fh, asynch_read_if) ;
```

and `asynch_read_if` could look like this

```
static void
asynch_read_if(fh, buffer)
int fh ;
char *      buffer ;
{
    dSP ;
    SV ** sv ;

    /* Get the callback associated with fh */
    sv =  hv_fetch(Mapping, (char*)&fh , sizeof(fh), FALSE) ;
    if (sv == (SV**)NULL)
        croak("Internal error...\n") ;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSViv(fh))) ;
    XPUSHs(sv_2mortal(newSVpv(buffer, 0))) ;
    PUTBACK ;

    /* Call the Perl sub */
    perl_call_sv(*sv, G_DISCARD) ;
}
```

For completeness, here is `asynch_close`. This shows how to remove the entry from the hash `Mapping`.

```
void
asynch_close(fh)
    int     fh
    CODE:
    /* Remove the entry from the hash */
    (void) hv_delete(Mapping, (char*)&fh, sizeof(fh), G_DISCARD) ;

    /* Now call the real asynch_close */
    asynch_close(fh) ;
```

So the Perl interface would look like this

```
sub callback1
{
    my($handle, $buffer) = @_ ;
}

# Register the Perl callback
asynch_read($fh, \&callback1) ;

asynch_close($fh) ;
```

The mapping between the C callback and Perl is stored in the global hash `Mapping` this time. Using a hash has the distinct advantage that it allows an unlimited number of callbacks to be registered.

What if the interface provided by the C callback doesn't contain a parameter which allows the file handle to Perl subroutine mapping? Say in the asynchronous i/o package, the callback function gets passed only the `buffer` parameter like this

```
void
ProcessRead(buffer)
char *      buffer ;
{
    ...
}
```

Without the file handle there is no straightforward way to map from the C callback to the Perl subroutine.

In this case a possible way around this problem is to predefine a series of C functions to act as the interface to Perl, thus

```
#define MAX_CB              3
#define NULL_HANDLE -1
typedef void (*FnMap)() ;

struct MapStruct {
    FnMap    Function ;
    SV *     PerlSub ;
    int      Handle ;
  } ;

static void  fn1() ;
static void  fn2() ;
static void  fn3() ;

static struct MapStruct Map [MAX_CB] =
    {
        { fn1, NULL, NULL_HANDLE },
        { fn2, NULL, NULL_HANDLE },
        { fn3, NULL, NULL_HANDLE }
    } ;

static void
Pcb(index, buffer)
int index ;
char * buffer ;
{
    dSP ;
```

```
        PUSHMARK(SP) ;
        XPUSHs(sv_2mortal(newSVpv(buffer, 0))) ;
        PUTBACK ;

        /* Call the Perl sub */
        perl_call_sv(Map[index].PerlSub, G_DISCARD) ;
    }

    static void
    fn1(buffer)
    char * buffer ;
    {
        Pcb(0, buffer) ;
    }

    static void
    fn2(buffer)
    char * buffer ;
    {
        Pcb(1, buffer) ;
    }

    static void
    fn3(buffer)
    char * buffer ;
    {
        Pcb(2, buffer) ;
    }

    void
    array_asynch_read(fh, callback)
        int             fh
        SV *    callback
        CODE:
        int index ;
        int null_index = MAX_CB ;

        /* Find the same handle or an empty entry */
        for (index = 0 ; index < MAX_CB ; ++index)
        {
            if (Map[index].Handle == fh)
                break ;

            if (Map[index].Handle == NULL_HANDLE)
                null_index = index ;
        }

        if (index == MAX_CB && null_index == MAX_CB)
            croak ("Too many callback functions registered\n") ;

        if (index == MAX_CB)
            index = null_index ;
```

```
        /* Save the file handle */
        Map[index].Handle = fh ;

        /* Remember the Perl sub */
        if (Map[index].PerlSub == (SV*)NULL)
            Map[index].PerlSub = newSVsv(callback) ;
        else
            SvSetSV(Map[index].PerlSub, callback) ;

        asynch_read(fh, Map[index].Function) ;

    void
    array_asynch_close(fh)
        int     fh
        CODE:
        int index ;

        /* Find the file handle */
        for (index = 0; index < MAX_CB ; ++ index)
            if (Map[index].Handle == fh)
                break ;

        if (index == MAX_CB)
            croak ("could not close fh %d\n", fh) ;

        Map[index].Handle = NULL_HANDLE ;
        SvREFCNT_dec(Map[index].PerlSub) ;
        Map[index].PerlSub = (SV*)NULL ;

        asynch_close(fh) ;
```

In this case the functions `fn1`, `fn2`, and `fn3` are used to remember the Perl subroutine to be called. Each of the functions holds a separate hard-wired index which is used in the function `Pcb` to access the `Map` array and actually call the Perl subroutine.

There are some obvious disadvantages with this technique.

Firstly, the code is considerably more complex than with the previous example.

Secondly, there is a hard-wired limit (in this case 3) to the number of callbacks that can exist simultaneously. The only way to increase the limit is by modifying the code to add more functions and then recompiling. None the less, as long as the number of functions is chosen with some care, it is still a workable solution and in some cases is the only one available.

To summarize, here are a number of possible methods for you to consider for storing the mapping between C and the Perl callback

1. Ignore the problem – Allow only 1 callback
   For a lot of situations, like interfacing to an error handler, this may be a perfectly adequate solution.

2. Create a sequence of callbacks – hard wired limit
   If it is impossible to tell from the parameters passed back from the C callback what the context is, then you may need to create a sequence of C callback interface functions, and store pointers to each in an array.

3. Use a parameter to map to the Perl callback
   A hash is an ideal mechanism to store the mapping between C and Perl.

### Alternate Stack Manipulation

Although I have made use of only the `POP*` macros to access values returned from Perl subroutines, it is also possible to bypass these macros and read the stack using the `ST` macro (See the *perlxs* manpage for a full description of the `ST` macro).

Most of the time the `POP*` macros should be adequate, the main problem with them is that they force you to process the returned values in sequence. This may not be the most suitable way to process the values in some cases. What we want is to be able to access the stack in a random order. The `ST` macro as used when coding an XSUB is ideal for this purpose.

The code below is the example given in the section *Returning a list of values* recoded to use `ST` instead of `POP*`.

```
static void
call_AddSubtract2(a, b)
int a ;
int b ;
{
    dSP ;
    I32 ax ;
    int count ;

    ENTER ;
    SAVETMPS;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK ;

    count = perl_call_pv("AddSubtract", G_ARRAY);

    SPAGAIN ;
    SP -= count ;
    ax = (SP - PL_stack_base) + 1 ;

    if (count != 2)
        croak("Big trouble\n") ;

    printf ("%d + %d = %d\n", a, b, SvIV(ST(0))) ;
    printf ("%d - %d = %d\n", a, b, SvIV(ST(1))) ;

    PUTBACK ;
    FREETMPS ;
    LEAVE ;
}
```

Notes

1. Notice that it was necessary to define the variable `ax`. This is because the `ST` macro expects it to exist. If we were in an XSUB it would not be necessary to define `ax` as it is already defined for you.

2. The code

```
SPAGAIN ;
SP -= count ;
ax = (SP - PL_stack_base) + 1 ;
```

sets the stack up so that we can use the `ST` macro.

3. Unlike the original coding of this example, the returned values are not accessed in reverse order. So `ST(0)` refers to the first value returned by the Perl subroutine and `ST(count-1)` refers to the last.

### Creating and calling an anonymous subroutine in C

As we've already shown, `perl_call_sv` can be used to invoke an anonymous subroutine. However, our example showed how Perl script invoking an XSUB to preform this operation. Let's see how it can be done inside our C code:

```
...

SV *cvrv = perl_eval_pv("sub { print 'You will not find me cluttering any namespace

...

perl_call_sv(cvrv, G_VOID|G_NOARGS);
```

`perl_eval_pv` is used to compile the anonymous subroutine, which will be the return value as well (read more about `perl_eval_pv` in the `perl_eval_pv` entry in the *perlguts* manpage). Once this code reference is in hand, it can be mixed in with all the previous examples we've shown.

## SEE ALSO

the *perlxs* manpage, the *perlguts* manpage, the *perlembed* manpage

## AUTHOR

Paul Marquess <*pmarquess@bfsec.bt.co.uk*>

Special thanks to the following people who assisted in the creation of the document.

Jeff Okamoto, Tim Bunce, Nick Gianniotis, Steve Kelem, Gurusamy Sarathy and Larry Wall.

## DATE

Version 1.3, 14th Apr 1997

**NAME**

perlhist − the Perl history records

**DESCRIPTION**

This document aims to record the Perl source code releases.

**INTRODUCTION**

Perl history in brief, by Larry Wall:

```
Perl 0 introduced Perl to my officemates.
Perl 1 introduced Perl to the world, and changed /\(...\|...\)/ to
    /(...|...)/.  \(Dan Faigin still hasn't forgiven me. :-\)
Perl 2 introduced Henry Spencer's regular expression package.
Perl 3 introduced the ability to handle binary data (embedded nulls).
Perl 4 introduced the first Camel book.  Really.  We mostly just
    switched version numbers so the book could refer to 4.000.
Perl 5 introduced everything else, including the ability to
    introduce everything else.
```

**THE KEEPERS OF THE PUMPKIN**

Larry Wall, Andy Dougherty, Tom Christiansen, Charles Bailey, Nick Ing-Simmons, Chip Salzenberg, Tim Bunce, Malcolm Beattie, Gurusamy Sarathy.

**PUMPKIN?**

[from Porting/pumpkin.pod in the Perl source code distribution]

Chip Salzenberg gets credit for that, with a nod to his cow orker, David Croy. We had passed around various names (baton, token, hot potato) but none caught on. Then, Chip asked:

[begin quote]

```
Who has the patch pumpkin?
```

To explain: David Croy once told me once that at a previous job, there was one tape drive and multiple systems that used it for backups. But instead of some high-tech exclusion software, they used a low-tech method to prevent multiple simultaneous backups: a stuffed pumpkin. No one was allowed to make backups unless they had the "backup pumpkin".

[end quote]

The name has stuck. The holder of the pumpkin is sometimes called the pumpking (keeping the source afloat?) or the pumpkineer (pulling the strings?).

**THE RECORDS**

```
Pump-  Release         Date            Notes
king                                   (by no means
                                        comprehensive,
                                        see Changes*
                                        for details)
===========================================================================

Larry   0              Classified.     Don't ask.
```

```
Larry   1.000          1987-Dec-18

         1.001..10     1988-Jan-30
         1.011..14     1988-Feb-02

Larry   2.000          1988-Jun-05

         2.001         1988-Jun-28

Larry   3.000          1989-Oct-18

         3.001         1989-Oct-26
         3.002..4      1989-Nov-11
         3.005         1989-Nov-18
         3.006..8      1989-Dec-22
         3.009..13     1990-Mar-02
         3.014         1990-Mar-13
         3.015         1990-Mar-14
         3.016..18     1990-Mar-28
         3.019..27     1990-Aug-10     User subs.
         3.028         1990-Aug-14
         3.029..36     1990-Oct-17
         3.037         1990-Oct-20
         3.040         1990-Nov-10
         3.041         1990-Nov-13
         3.042..43     1990-Jan-??
         3.044         1991-Jan-12

Larry   4.000          1991-Mar-21

         4.001..3      1991-Apr-12
         4.004..9      1991-Jun-07
         4.010         1991-Jun-10
         4.011..18     1991-Nov-05
         4.019         1991-Nov-11     Stable.
         4.020..33     1992-Jun-08
         4.034         1992-Jun-11
         4.035         1992-Jun-23
Larry    4.036         1993-Feb-05     Very stable.

         5.000alpha1   1993-Jul-31
         5.000alpha2   1993-Aug-16
         5.000alpha3   1993-Oct-10
         5.000alpha4   1993-???-??
         5.000alpha5   1993-???-??
         5.000alpha6   1994-Mar-18
         5.003alpha7   1994-Mar-25
Andy     5.000alpha8   1994-Apr-04
Larry    5.000alpha9   1994-May-05     ext appears.
         5.000alpha10  1994-???-??
         5.000alpha11  1994-???-??
Andy     5.000a11a     1994-Jul-07     To fit 14.
         5.000a11b     1994-Jul-14
         5.000a11c     1994-Jul-19
```

```
         5.000a11d     1994-Jul-22
Larry    5.000alpha12  1994-???-??
Andy     5.000a12a     1994-Aug-08
         5.000a12b     1994-Aug-15
         5.000a12c     1994-Aug-22
         5.000a12d     1994-Aug-22
         5.000a12e     1994-Aug-22
         5.000a12f     1994-Aug-24
         5.000a12g     1994-Aug-24
         5.000a12h     1994-Aug-24
Larry    5.000beta1    1994-???-??
Andy     5.000b1a      1994-???-??
Larry    5.000beta2    1994-Sep-14     Core slushified.
Andy     5.000b2a      1994-Sep-14
         5.000b2b      1994-Sep-17
         5.000b2c      1994-Sep-17
Larry    5.000beta3    1994-Sep-??
Andy     5.000b3a      1994-Sep-18
         5.000b3b      1994-Sep-22
         5.000b3c      1994-Sep-23
         5.000b3d      1994-Sep-27
         5.000b3e      1994-Sep-28
         5.000b3f      1994-Sep-30
         5.000b3g      1994-Oct-04
Andy     5.000b3h      1994-Oct-07

Larry   5.000          1994-Oct-18

Andy     5.000a        1994-Dec-19
         5.000b        1995-Jan-18
         5.000c        1995-Jan-18
         5.000d        1995-Jan-18
         5.000e        1995-Jan-18
         5.000f        1995-Jan-18
         5.000g        1995-Jan-18
         5.000h        1995-Jan-18
         5.000i        1995-Jan-26
         5.000j        1995-Feb-07
         5.000k        1995-Feb-11
         5.000l        1995-Feb-21
         5.000m        1995-???-??
         5.000n        1995-Mar-07

Larry   5.001          1995-Mar-13

Andy     5.001a        1995-Mar-15
         5.001b        1995-Mar-31
         5.001c        1995-Apr-07
         5.001d        1995-Apr-14
         5.001e        1995-Apr-18     Stable.
         5.001f        1995-May-31
         5.001g        1995-May-25
         5.001h        1995-May-25
         5.001i        1995-May-30
```

```
          5.001j        1995-Jun-05
          5.001k        1995-Jun-06
          5.001l        1995-Jun-06     Stable.
          5.001m        1995-Jul-02     Very stable.
          5.001n        1995-Oct-31     Very unstable.
          5.002beta1    1995-Nov-21
          5.002b1a      1995-Nov-??
          5.002b1b      1995-Dec-04
          5.002b1c      1995-Dec-04
          5.002b1d      1995-Dec-04
          5.002b1e      1995-Dec-08
          5.002b1f      1995-Dec-08
 Tom      5.002b1g      1995-Dec-21     Doc release.
 Andy     5.002b1h      1996-Jan-05
          5.002b2       1996-Jan-14
 Larry    5.002b3       1996-Feb-02
 Andy     5.002gamma    1996-Feb-11
 Larry    5.002delta    1996-Feb-27

 Larry   5.002          1996-Feb-29

 Charles  5.002_01      1996-Mar-25

         5.003          1996-Jun-25     Security release.

          5.003_01      1996-Jul-31
 Nick     5.003_02      1996-Aug-10
 Andy     5.003_03      1996-Aug-28
          5.003_04      1996-Sep-02
          5.003_05      1996-Sep-12
          5.003_06      1996-Oct-07
          5.003_07      1996-Oct-10
 Chip     5.003_08      1996-Nov-19
          5.003_09      1996-Nov-26
          5.003_10      1996-Nov-29
          5.003_11      1996-Dec-06
          5.003_12      1996-Dec-19
          5.003_13      1996-Dec-20
          5.003_14      1996-Dec-23
          5.003_15      1996-Dec-23
          5.003_16      1996-Dec-24
          5.003_17      1996-Dec-27
          5.003_18      1996-Dec-31
          5.003_19      1997-Jan-04
          5.003_20      1997-Jan-07
          5.003_21      1997-Jan-15
          5.003_22      1997-Jan-16
          5.003_23      1997-Jan-25
          5.003_24      1997-Jan-29
          5.003_25      1997-Feb-04
          5.003_26      1997-Feb-10
          5.003_27      1997-Feb-18
          5.003_28      1997-Feb-21
          5.003_90      1997-Feb-25     Ramping up to the 5.004 release.
```

```
           5.003_91      1997-Mar-01
           5.003_92      1997-Mar-06
           5.003_93      1997-Mar-10
           5.003_94      1997-Mar-22
           5.003_95      1997-Mar-25
           5.003_96      1997-Apr-01
           5.003_97      1997-Apr-03     Fairly widely used.
           5.003_97a     1997-Apr-05
           5.003_97b     1997-Apr-08
           5.003_97c     1997-Apr-10
           5.003_97d     1997-Apr-13
           5.003_97e     1997-Apr-15
           5.003_97f     1997-Apr-17
           5.003_97g     1997-Apr-18
           5.003_97h     1997-Apr-24
           5.003_97i     1997-Apr-25
           5.003_97j     1997-Apr-28
           5.003_98      1997-Apr-30
           5.003_99      1997-May-01
           5.003_99a     1997-May-09
           p54rc1        1997-May-12     Release Candidates.
           p54rc2        1997-May-14

  Chip    5.004          1997-May-15     A major maintenance release.

  Tim      5.004_01      1997-Jun-13     The 5.004 maintenance track.
           5.004_02      1997-Aug-07
           5.004_03      1997-Sep-05
           5.004_04      1997-Oct-15
           5.004m5t1     1998-Mar-04     Maintenance Trials (for 5.004_05).
           5.004_04-m2   1997-May-01
           5.004_04-m3   1998-May-15

  Malcolm  5.004_50      1997-Sep-09     The 5.005 development track.
           5.004_51      1997-Oct-02
           5.004_52      1997-Oct-15
           5.004_53      1997-Oct-16
           5.004_54      1997-Nov-14
           5.004_55      1997-Nov-25
           5.004_56      1997-Dec-18
           5.004_57      1998-Feb-03
           5.004_58      1998-Feb-06
           5.004_59      1998-Feb-13
           5.004_60      1998-Feb-20
           5.004_61      1998-Feb-27
           5.004_62      1998-Mar-06
           5.004_63      1998-Mar-17
           5.004_64      1998-Apr-03
           5.004_65      1998-May-15
           5.004_66      1998-May-29
  Sarathy  5.004_67      1998-Jun-15
           5.004_68      1998-Jun-23
           5.004_69      1998-Jun-29
           5.004_70      1998-Jul-06
```

```
5.004_71      1998-Jul-09
5.004_72      1998-Jul-12
5.004_73      1998-Jul-13
5.004_74      1998-Jul-14
5.004_75      1998-Jul-15
5.004_76      1998-Jul-21
5.005         1998-Jul-22
```

**SELECTED RELEASE SIZES**

For example the notation "core: 212 29" in the release 1.000 means that it had in the core 212 kilobytes, in 29 files. The "core".."doc" are explained below.

```
release         core       lib         ext        t          doc
======================================================================

1.000            212  29      -   -      -   -      38  51      62   3
1.014            219  29      -   -      -   -      39  52      68   4
2.000            309  31      2   3      -   -      55  57      92   4
2.001            312  31      2   3      -   -      55  57      94   4
3.000            508  36     24  11      -   -      79  73     156   5
3.044            645  37     61  20      -   -      90  74     190   6
4.000            635  37     59  20      -   -      91  75     198   4
4.019            680  37     85  29      -   -      98  76     199   4
4.036            709  37     89  30      -   -      98  76     208   5
5.000alpha2      785  50    114  32      -   -     112  86     209   5
5.000alpha3      801  50    117  33      -   -     121  87     209   5
5.000alpha9     1022  56    149  43    116  29     125  90     217   6
5.000a12h        978  49    140  49    205  46     152  97     228   9
5.000b3h        1035  53    232  70    216  38     162  94     218  21
5.000           1038  53    250  76    216  38     154  92     536  62
5.001m          1071  54    388  82    240  38     159  95     544  29
5.002           1121  54    661 101    287  43     155  94     847  35
5.003           1129  54    680 102    291  43     166 100     853  35
5.003_07        1231  60    748 106    396  53     213 137     976  39
5.004           1351  60   1230 136    408  51     355 161    1587  55
5.004_01        1356  60   1258 138    410  51     358 161    1587  55
5.004_04        1375  60   1294 139    413  51     394 162    1629  55
5.004_51        1401  61   1260 140    413  53     358 162    1594  56
5.004_53        1422  62   1295 141    438  70     394 162    1637  56
5.004_56        1501  66   1301 140    447  74     408 165    1648  57
5.004_59        1555  72   1317 142    448  74     424 171    1678  58
5.004_62        1602  77   1327 144    629  92     428 173    1674  58
5.004_65        1626  77   1358 146    615  92     446 179    1698  60
```

The "core"..."doc" mean the following files from the Perl source code distribution. The glob notation ** means recursively, (.) means regular files.

```
core   *.[hcy]
lib    lib/**/*.p[ml]
ext    ext/**/*.{[hcyt],xs,pm}
t      t/**/*(.)
doc    {README*,INSTALL,*[_.]man{,.?},pod/**/*.pod}
```

Here are some statistics for the other subdirectories and one file in the Perl source distribution for somewhat more selected releases.

```
======================================================================
  Legend:  kB   #

           1.014   2.001   3.044   4.000   4.019   4.036

atarist      -  -    -  -    -  -    -  -    -  -  113 31
Configure   31  1   37  1   62  1   73  1   83  1   86  1
eg           -  -   34 28   47 39   47 39   47 39   47 39
emacs        -  -    -  -    -  -   67  4   67  4   67  4
h2pl         -  -    -  -   12 12   12 12   12 12   12 12
hints        -  -    -  -    -  -    -  -    5 42   11 56
msdos        -  -    -  -   41 13   57 15   58 15   60 15
os2          -  -    -  -   63 22   81 29   81 29  113 31
usub         -  -    -  -   21 16   25  7   43  8   43  8
x2p        103 17  104 17  137 17  147 18  152 19  154 19

======================================================================

           5.000a2 5.000a12h 5.000b3h 5.000  5.001m  5.002   5.003

atarist    113 31  113 31    -  -      -  -    -  -    -  -    -  -
bench        -  -    0  1    -  -      -  -    -  -    -  -    -  -
Bugs         2  5   26  1    -  -      -  -    -  -    -  -    -  -
dlperl      40  5    -  -    -  -      -  -    -  -    -  -    -  -
do         127 71    -  -    -  -      -  -    -  -    -  -    -  -
Configure    -  -  153  1  159  1    160  1  180  1  201  1  201  1
Doc          -  -   26  1   75  7     11  1   11  1    -  -    -  -
eg          79 58   53 44   51 43     54 44   54 44   54 44   54 44
emacs       67  4  104  6  104  6    104  1  104  6  108  1  108  1
h2pl        12 12   12 12   12 12     12 12   12 12   12 12   12 12
hints       11 56   12 46   18 48     18 48   44 56   73 59   77 60
msdos       60 15   60 15    -  -      -  -    -  -    -  -    -  -
os2        113 31  113 31    -  -      -  -    -  -   84 17   56 10
U            -  -   62  8  112 42      -  -    -  -    -  -    -  -
usub        43  8    -  -    -  -      -  -    -  -    -  -    -  -
utils        -  -    -  -    -  -      -  -    -  -   87  7   88  7
vms          -  -   80  7  123  9    184 15  304 20  500 24  475 26
x2p        171 22  171 21  162 20    162 20  279 20  280 20  280 20

======================================================================
```

| | 5.003_07 | | 5.004 | | 5.004_04 | | 5.004_62 | | 5.004_65 | |
|---|---|---|---|---|---|---|---|---|---|---|
| beos | – | – | – | – | – | – | – | – | 1 | 1 |
| Configure | 217 | 1 | 225 | 1 | 225 | 1 | 240 | 1 | 248 | 1 |
| cygwin32 | – | – | 23 | 5 | 23 | 5 | 23 | 5 | 24 | 5 |
| djgpp | – | – | – | – | – | – | 14 | 5 | 14 | 5 |
| eg | 54 | 44 | 81 | 62 | 81 | 62 | 81 | 62 | 81 | 62 |
| emacs | 143 | 1 | 194 | 1 | 204 | 1 | 212 | 2 | 212 | 2 |
| h2pl | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| hints | 90 | 62 | 129 | 69 | 132 | 71 | 144 | 72 | 151 | 74 |
| os2 | 117 | 42 | 121 | 42 | 127 | 42 | 127 | 44 | 129 | 44 |
| plan9 | 79 | 15 | 82 | 15 | 82 | 15 | 82 | 15 | 82 | 15 |
| Porting | 51 | 1 | 94 | 2 | 109 | 4 | 203 | 6 | 234 | 8 |
| qnx | – | – | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 |
| utils | 97 | 7 | 112 | 8 | 118 | 8 | 124 | 8 | 156 | 9 |
| vms | 505 | 27 | 518 | 34 | 524 | 34 | 538 | 34 | 569 | 34 |
| win32 | – | – | 285 | 33 | 378 | 36 | 470 | 39 | 493 | 39 |
| x2p | 280 | 19 | 281 | 19 | 281 | 19 | 281 | 19 | 282 | 19 |

## SELECTED PATCH SIZES

The "diff lines kb" means that for example the patch 5.003_08, to be applied on top of the 5.003_07 (or whatever was before the 5.003_08) added lines for 110 kilobytes, it removed lines for 19 kilobytes, and changed lines for 424 kilobytes. Just the lines themselves are counted, not their context. The "+ – !" become from the *diff*(1)s context diff output format.

```
Pump-  Release          Date           diff lines kB
king                                      +   -   !
===========================================================================
```

```
 Chip     5.003_08       1996-Nov-19     110  19 424
          5.003_09       1996-Nov-26      38   9 248
          5.003_10       1996-Nov-29      29   2  27
          5.003_11       1996-Dec-06      73  12 165
          5.003_12       1996-Dec-19     275   6 436
          5.003_13       1996-Dec-20      95   1  56
          5.003_14       1996-Dec-23      23   7 333
          5.003_15       1996-Dec-23       0   0   1
          5.003_16       1996-Dec-24      12   3  50
          5.003_17       1996-Dec-27      19   1  14
          5.003_18       1996-Dec-31      21   1  32
          5.003_19       1997-Jan-04      80   3  85
          5.003_20       1997-Jan-07      18   1 146
          5.003_21       1997-Jan-15      38  10 221
          5.003_22       1997-Jan-16       4   0  18
          5.003_23       1997-Jan-25      71  15 119
          5.003_24       1997-Jan-29     426   1  20
          5.003_25       1997-Feb-04      21   8 169
          5.003_26       1997-Feb-10      16   1  15
          5.003_27       1997-Feb-18      32  10  38
          5.003_28       1997-Feb-21      58   4  66
          5.003_90       1997-Feb-25      22   2  34
          5.003_91       1997-Mar-01      37   1  39
          5.003_92       1997-Mar-06      16   3  69
          5.003_93       1997-Mar-10      12   3  15
          5.003_94       1997-Mar-22     407   7 200
          5.003_95       1997-Mar-25      41   1  37
          5.003_96       1997-Apr-01     283   5 261
          5.003_97       1997-Apr-03      13   2  34
          5.003_97a      1997-Apr-05      57   1  27
          5.003_97b      1997-Apr-08      14   1  20
          5.003_97c      1997-Apr-10      20   1  16
          5.003_97d      1997-Apr-13       8   0  16
          5.003_97e      1997-Apr-15      15   4  46
          5.003_97f      1997-Apr-17       7   1  33
          5.003_97g      1997-Apr-18       6   1  42
          5.003_97h      1997-Apr-24      23   3  68
          5.003_97i      1997-Apr-25      23   1  31
          5.003_97j      1997-Apr-28      36   1  49
          5.003_98       1997-Apr-30     171  12 539
          5.003_99       1997-May-01       6   0   7
          5.003_99a      1997-May-09      36   2  61
          p54rc1         1997-May-12       8   1  11
          p54rc2         1997-May-14       6   0  40

        5.004            1997-May-15       4   0   4

 Tim      5.004_01       1997-Jun-13     222  14  57
          5.004_02       1997-Aug-07     112  16 119
          5.004_03       1997-Sep-05     109   0  17
          5.004_04       1997-Oct-15      66   8 173
```

**THE KEEPERS OF THE RECORDS**

Jarkko Hietaniemi <*jhi@iki.fi*>.

Thanks to the collective memory of the Perlfolk. In addition to the Keepers of the Pumpkin also Alan Champion, Andreas König, John Macdonald, Matthias Neeracher, Michael Peppler, Randal Schwartz, and Paul D. Smith sent corrections and additions.

## NAME

perldelta – what's new for perl5.005

## DESCRIPTION

This document describes differences between the 5.004 release and this one.

### About the new versioning system

Perl is now developed on two tracks: a maintenance track that makes small, safe updates to released production versions with emphasis on compatibility; and a development track that pursues more aggressive evolution. Maintenance releases (which should be considered production quality) have subversion numbers that run from `1` to `49`, and development releases (which should be considered "alpha" quality) run from `50` to `99`.

Perl 5.005 is the combined product of the new dual-track development scheme.

### Incompatible Changes

**WARNING: This version is not binary compatible with Perl 5.004.**

Starting with Perl 5.004_50 there were many deep and far-reaching changes to the language internals. If you have dynamically loaded extensions that you built under perl 5.003 or 5.004, you can continue to use them with 5.004, but you will need to rebuild and reinstall those extensions to use them 5.005. See the *INSTALL* manpage for detailed instructions on how to upgrade.

#### Default installation structure has changed

The new Configure defaults are designed to allow a smooth upgrade from 5.004 to 5.005, but you should read the *INSTALL* manpage for a detailed discussion of the changes in order to adapt them to your system.

#### Perl Source Compatibility

When none of the experimental features are enabled, there should be very few user-visible Perl source compatibility issues.

If threads are enabled, then some caveats apply. `@_` and `$_` become lexical variables. The effect of this should be largely transparent to the user, but there are some boundary conditions under which user will need to be aware of the issues. For example, `local(@_)` results in a "Can't localize lexical variable `@_` ..." message. This may be enabled in a future version.

Some new keywords have been introduced. These are generally expected to have very little impact on compatibility. See the section on *New* `INIT` keyword, the section on *New* `lock` keyword, and the section on *New* `qr//` operator.

Certain barewords are now reserved. Use of these will provoke a warning if you have asked for them with the `-w` switch. See the `our` manpage is now a reserved word.

#### C Source Compatibility

There have been a large number of changes in the internals to support the new features in this release.

Core sources now require ANSI C compiler

An ANSI C compiler is now **required** to build perl. See *INSTALL*.

All Perl global variables must now be referenced with an explicit prefix

All Perl global variables that are visible for use by extensions now have a `PL_` prefix. New extensions should `not` refer to perl globals by their unqualified names. To preserve sanity, we provide limited backward compatibility for globals that are being widely used like `sv_undef` and `na` (which should now be written as `PL_sv_undef`, `PL_na` etc.)

If you find that your XS extension does not compile anymore because a perl global is not visible, try adding a `PL_` prefix to the global and rebuild.

It is strongly recommended that all functions in the Perl API that don't begin with `perl` be referenced with a `Perl_` prefix. The bare function names without the `Perl_` prefix are supported with macros, but this support may cease in a future release.

See the section on *API LISTING* in the *perlguts* manpage.

Enabling threads has source compatibility issues

Perl built with threading enabled requires extensions to use the new `dTHR` macro to initialize the handle to access per-thread data. If you see a compiler error that talks about the variable `thr` not being declared (when building a module that has XS code), you need to add `dTHR;` at the beginning of the block that elicited the error.

The API function `perl_get_sv("@",FALSE)` should be used instead of directly accessing perl globals as `GvSV(errgv)`. The API call is backward compatible with existing perls and provides source compatibility with threading is enabled.

See the section on *API Changes for more information*.

### Binary Compatibility

This version is NOT binary compatible with older versions. All extensions will need to be recompiled. Further binaries built with threads enabled are incompatible with binaries built without. This should largely be transparent to the user, as all binary incompatible configurations have their own unique architecture name, and extension binaries get installed at unique locations. This allows coexistence of several configurations in the same directory hierarchy. See *INSTALL*.

### Security fixes may affect compatibility

A few taint leaks and taint omissions have been corrected. This may lead to "failure" of scripts that used to work with older versions. Compiling with −DINCOMPLETE_TAINTS provides a perl with minimal amounts of changes to the tainting behavior. But note that the resulting perl will have known insecurities.

Oneliners with the `-e` switch do not create temporary files anymore.

### Relaxed new mandatory warnings introduced in 5.004

Many new warnings that were introduced in 5.004 have been made optional. Some of these warnings are still present, but perl's new features make them less often a problem. See the section on *New Diagnostics*.

### Licensing

Perl has a new Social Contract for contributors. See *Porting/Contract*.

The license included in much of the Perl documentation has changed. Most of the Perl documentation was previously under the implicit GNU General Public License or the Artistic License (at the user's choice). Now much of the documentation unambigously states the terms under which it may be distributed. Those terms are in general much less restrictive than the GNU GPL. See the *perl* manpage and the individual perl man pages listed therein.

## Core Changes

### Threads

WARNING: Threading is considered an **experimental** feature. Details of the implementation may change without notice. There are known limitations and some bugs. These are expected to be fixed in future versions.

See the *README.threads* manpage.

### Compiler

WARNING: The Compiler and related tools are considered **experimental**. Features may change without notice, and there are known limitations and bugs. Since the compiler is fully external to perl, the default configuration will build and install it.

The Compiler produces three different types of transformations of a perl program. The C backend generates C code that captures perl's state just before execution begins. It eliminates the compile-time overheads of the regular perl interpreter, but the run-time performance remains comparatively the same. The CC backend generates optimized C code equivalent to the code path at run-time. The CC backend has greater potential for big optimizations, but only a few optimizations are implemented currently. The Bytecode backend generates a platform independent bytecode representation of the interpreter's state just before execution. Thus, the Bytecode back end also eliminates much of the compilation overhead of the interpreter.

The compiler comes with several valuable utilities.

`B::Lint` is an experimental module to detect and warn about suspicious code, especially the cases that the `-w` switch does not detect.

`B::Deparse` can be used to demystify perl code, and understand how perl optimizes certain constructs.

`B::Xref` generates cross reference reports of all definition and use of variables, subroutines and formats in a program.

`B::Showlex` show the lexical variables used by a subroutine or file at a glance.

`perlcc` is a simple frontend for compiling perl.

See `ext/B/README`, the section on *B*, and the respective compiler modules.

**Regular Expressions**

Perl's regular expression engine has been seriously overhauled, and many new constructs are supported. Several bugs have been fixed.

Here is an itemized summary:

Many new and improved optimizations
  Changes in the RE engine:

```
Unneeded nodes removed;
Substrings merged together;
New types of nodes to process (SUBEXPR)* and similar expressions
    quickly, used if the SUBEXPR has no side effects and matches
    strings of the same length;
Better optimizations by lookup for constant substrings;
Better search for constants substrings anchored by $ ;
```

  Changes in Perl code using RE engine:

```
More optimizations to s/longer/short/;
study() was not working;
/blah/ may be optimized to an analogue of index() if $& $` $' not seen;
Unneeded copying of matched-against string removed;
Only matched part of the string is copying if $` $' were not seen;
```

Many bug fixes
  Note that only the major bug fixes are listed here. See *Changes* for others.

```
Backtracking might not restore start of $3.
No feedback if max count for * or + on "complex" subexpression
    was reached, similarly (but at compile time) for {3,34567}
Primitive restrictions on max count introduced to decrease a
    possibility of a segfault;
(ZERO-LENGTH)* could segfault;
(ZERO-LENGTH)* was prohibited;
Long REs were not allowed;
/RE/g could skip matches at the same position after a
  zero-length match;
```

New regular expression constructs
The following new syntax elements are supported:

```
(?<=RE)
(?<!RE)
(?{ CODE })
(?i-x)
(?i:RE)
(?(COND)YES_RE|NO_RE)
(?>RE)
\z
```

New operator for precompiled regular expressions
See the section on *New* `qr//` operator.

Other improvements

```
Better debugging output (possibly with colors),
    even from non-debugging Perl;
RE engine code now looks like C, not like assembler;
Behaviour of RE modifiable by `use re' directive;
Improved documentation;
Test suite significantly extended;
Syntax [:^upper:] etc., reserved inside character classes;
```

Incompatible changes

```
(?i) localized inside enclosing group;
$( is not interpolated into RE any more;
/RE/g may match at the same position (with non-zero length)
    after a zero-length match (bug fix).
```

See the *perlre* manpage and the *perlop* manpage.

### Improved *malloc()*

See banner at the beginning of `malloc.c` for details.

### Quicksort is internally implemented

Perl now contains its own highly optimized *qsort()* routine. The new *qsort()* is resistant to inconsistent comparison functions, so Perl's `sort()` will not provoke coredumps any more when given poorly written sort subroutines. (Some C library `qsort()`s that were being used before used to have this problem.) In our testing, the new `qsort()` required the minimal number of pair-wise compares on average, among all known `qsort()` implementations.

See `perlfunc/sort`.

### Reliable signals

Perl's signal handling is susceptible to random crashes, because signals arrive asynchronously, and the Perl runtime is not reentrant at arbitrary times.

However, one experimental implementation of reliable signals is available when threads are enabled. See `Thread::Signal`. Also see *INSTALL* for how to build a Perl capable of threads.

**Reliable stack pointers**

The internals now reallocate the perl stack only at predictable times. In particular, magic calls never trigger reallocations of the stack, because all reentrancy of the runtime is handled using a "stack of stacks". This should improve reliability of cached stack pointers in the internals and in XSUBs.

**More generous treatment of carriage returns**

Perl used to complain if it encountered literal carriage returns in scripts. Now they are mostly treated like whitespace within program text. Inside string literals and here documents, literal carriage returns are ignored if they occur paired with newlines, or get interpreted as newlines if they stand alone. This behavior means that literal carriage returns in files should be avoided. You can get the older, more compatible (but less generous) behavior by defining the preprocessor symbol `PERL_STRICT_CR` when building perl. Of course, all this has nothing whatever to do with how escapes like `\r` are handled within strings.

Note that this doesn't somehow magically allow you to keep all text files in DOS format. The generous treatment only applies to files that perl itself parses. If your C compiler doesn't allow carriage returns in files, you may still be unable to build modules that need a C compiler.

**Memory leaks**

`substr`, `pos` and `vec` don't leak memory anymore when used in lvalue context. Many small leaks that impacted applications that embed multiple interpreters have been fixed.

**Better support for multiple interpreters**

The build-time option `-DMULTIPLICITY` has had many of the details reworked. Some previously global variables that should have been per-interpreter now are. With care, this allows interpreters to call each other. See the `PerlInterp` extension on CPAN.

**Behavior of** *local()* on array and hash elements is now well-defined

See the section on *Temporary Values via local()* in the *perlsub* manpage.

`%!` is transparently tied to the the *Errno* manpage module

See the *perlvar* manpage, and the *Errno* manpage.

**Pseudo-hashes are supported**

See the *perlref* manpage.

`EXPR foreach EXPR` is supported

See the *perlsyn* manpage.

**Keywords can be globally overridden**

See the *perlsub* manpage.

`$^E` is meaningful on Win32

See the *perlvar* manpage.

`foreach (1..1000000)` optimized

`foreach (1..1000000)` is now optimized into a counting loop. It does not try to allocate a 1000000-size list anymore.

`Foo::` can be used as implicitly quoted package name

Barewords caused unintuitive behavior when a subroutine with the same name as a package happened to be defined. Thus, `new Foo @args`, use the result of the call to `Foo()` instead of `Foo` being treated as a literal. The recommended way to write barewords in the indirect object slot is `new Foo:: @args`. Note that the method `new()` is called with a first argument of `Foo`, not `Foo::` when you do that.

`exists $Foo::{Bar::}` tests existence of a package

It was impossible to test for the existence of a package without actually creating it before. Now `exists $Foo::{Bar::}` can be used to test if the `Foo::Bar` namespace has been created.

**Better locale support**

See the *perllocale* manpage.

**Experimental support for 64-bit platforms**

Perl5 has always had 64-bit support on systems with 64-bit longs. Starting with 5.005, the beginnings of experimental support for systems with 32-bit long and 64-bit 'long long' integers has been added. If you add −DUSE_LONG_LONG to your ccflags in config.sh (or manually define it in perl.h) then perl will be built with 'long long' support. There will be many compiler warnings, and the resultant perl may not work on all systems. There are many other issues related to third-party extensions and libraries. This option exists to allow people to work on those issues.

*prototype()* returns useful results on builtins

See the `prototype` entry in the *perlfunc* manpage.

**Extended support for exception handling**

`die()` now accepts a reference value, and `$@` gets set to that value in exception traps. This makes it possible to propagate exception objects. This is an undocumented **experimental** feature.

**Re-blessing in** *DESTROY()* supported for chaining *DESTROY()* methods

See the `Destructors` entry in the *perlobj* manpage.

**All** `printf` format conversions are handled internally

See the `printf` entry in the *perlfunc* manpage.

**New** `INIT` keyword

`INIT` subs are like `BEGIN` and `END`, but they get run just before the perl runtime begins execution. e.g., the Perl Compiler makes use of `INIT` blocks to initialize and resolve pointers to XSUBs.

**New** `lock` keyword

The `lock` keyword is the fundamental synchronization primitive in threaded perl. When threads are not enabled, it is currently a noop.

To minimize impact on source compatibility this keyword is "weak", i.e., any user-defined subroutine of the same name overrides it, unless a `use Thread` has been seen.

**New** `qr//` operator

The `qr//` operator, which is syntactically similar to the other quote-like operators, is used to create precompiled regular expressions. This compiled form can now be explicitly passed around in variables, and interpolated in other regular expressions. See the *perlop* manpage.

`our` is now a reserved word

Calling a subroutine with the name `our` will now provoke a warning when using the `-w` switch.

### Tied arrays are now fully supported

See the *Tie::Array* manpage.

### Tied handles support is better

Several missing hooks have been added. There is also a new base class for TIEARRAY implementations. See the *Tie::Array* manpage.

### 4th argument to substr

*substr()* can now both return and replace in one operation. The optional 4th argument is the replacement string. See the `substr` entry in the *perlfunc* manpage.

### Negative LENGTH argument to splice

*splice()* with a negative LENGTH argument now work similar to what the LENGTH did for *substr()*. Previously a negative LENGTH was treated as 0. See the `splice` entry in the *perlfunc* manpage.

### Magic lvalues are now more magical

When you say something like `substr($x, 5) = "hi"`, the scalar returned by *substr()* is special, in that any modifications to it affect `$x`. (This is called a 'magic lvalue' because an 'lvalue' is something on the left side of an assignment.) Normally, this is exactly what you would expect to happen, but Perl uses the same magic if you use *substr()*, *pos()*, or *vec()* in a context where they might be modified, like taking a reference with \ or as an argument to a sub that modifies `@_`. In previous versions, this 'magic' only went one way, but now changes to the scalar the magic refers to ($x in the above example) affect the magic lvalue too. For instance, this code now acts differently:

```
$x = "hello";
sub printit {
    $x = "g'bye";
    print $_[0], "\n";
}
printit(substr($x, 0, 5));
```

In previous versions, this would print "hello", but it now prints "g'bye".

### <> now reads in records

If `$/` is a referenence to an integer, or a scalar that holds an integer, <> will read in records instead of lines. For more info, see the section on *$/* in the *perlvar* manpage.

## Supported Platforms

Configure has many incremental improvements. Site-wide policy for building perl can now be made persistent, via Policy.sh. Configure also records the command-line arguments used in *config.sh*.

### New Platforms

BeOS is now supported. See the *README.beos* manpage.

DOS is now supported under the DJGPP tools. See the *README.dos* manpage.

MPE/iX is now supported. See the *README.mpeix* manpage.

MVS (OS390) is now supported. See the *README.os390* manpage.

### Changes in existing support

Win32 support has been vastly enhanced. Support for Perl Object, a C++ encapsulation of Perl. GCC and EGCS are now supported on Win32. See *README.win32*, aka the *perlwin32* manpage.

VMS configuration system has been rewritten. See the *README.vms* manpage.

The hints files for most Unix platforms have seen incremental improvements.

## Modules and Pragmata

### New Modules

B
: Perl compiler and tools. See the section on *B*.

Data::Dumper
: A module to pretty print Perl data. See the *Data::Dumper* manpage.

Errno
: A module to look up errors more conveniently. See the *Errno* manpage.

File::Spec
: A portable API for file operations.

ExtUtils::Installed
: Query and manage installed modules.

ExtUtils::Packlist
: Manipulate .packlist files.

Fatal
: Make functions/builtins succeed or die.

IPC::SysV
: Constants and other support infrastructure for System V IPC operations in perl.

Test
: A framework for writing testsuites.

Tie::Array
: Base class for tied arrays.

Tie::Handle
: Base class for tied handles.

Thread
: Perl thread creation, manipulation, and support.

attrs
: Set subroutine attributes.

fields
: Compile-time class fields.

re
: Various pragmata to control behavior of regular expressions.

### Changes in existing modules

CGI
: CGI has been updated to version 2.42.

POSIX
: POSIX now has its own platform-specific hints files.

DB_File
: DB_File supports version 2.x of Berkeley DB. See `ext/DB_File/Changes`.

MakeMaker
: MakeMaker now supports writing empty makefiles, provides a way to specify that site *umask()* policy should be honored. There is also better support for manipulation of .packlist files, and getting information about installed modules.

Extensions that have both architecture-dependent and architecture-independent files are now always installed completely in the architecture-dependent locations. Previously, the shareable parts were shared both across architectures and across perl versions and were therefore liable to be overwritten with newer versions that might have subtle incompatibilities.

CPAN
: See <perlmodinstall> and the *CPAN* manpage.

Cwd
: Cwd::cwd is faster on most platforms.

Benchmark
: Keeps better time.

**Utility Changes**

`h2ph` and related utilities have been vastly overhauled.

`perlcc`, a new experimental front end for the compiler is available.

The crude GNU `configure` emulator is now called `configure.gnu` to avoid trampling on `Configure` under case-insensitive filesystems.

`perldoc` used to be rather slow. The slower features are now optional. In particular, case-insensitive searches need the `-i` switch, and recursive searches need `-r`. You can set these switches in the `PERLDOC` environment variable to get the old behavior.

**Documentation Changes**

Config.pm now has a glossary of variables.

*Porting/patching.pod* has detailed instructions on how to create and submit patches for perl.

the *perlport* manpage specifies guidelines on how to write portably.

the *perlmodinstall* manpage describes how to fetch and install modules from `CPAN` sites.

Some more Perl traps are documented now. See the *perltrap* manpage.

**New Diagnostics**

Ambiguous call resolved as CORE::%s(), qualify as such or use &

(W) A subroutine you have declared has the same name as a Perl keyword, and you have used the name without qualification for calling one or the other. Perl decided to call the builtin because the subroutine is not imported.

To force interpretation as a subroutine call, either put an ampersand before the subroutine name, or qualify the name with its package. Alternatively, you can import the subroutine (or pretend that it's imported with the `use subs` pragma).

To silently interpret it as the Perl operator, use the `CORE::` prefix on the operator (e.g. `CORE::log($x)`) or by declaring the subroutine to be an object method (see the *attrs* manpage).

Bad index while coercing array into hash

(F) The index looked up in the hash found as the 0'th element of a pseudo-hash is not legal. Index values must be at 1 or greater. See the *perlref* manpage.

Bareword "%s" refers to nonexistent package

(W) You used a qualified bareword of the form `Foo::`, but the compiler saw no other uses of that namespace before that point. Perhaps you need to predeclare a package?

Can't call method "%s" on an undefined value

(F) You used the syntax of a method call, but the slot filled by the object reference or package name contains an undefined value. Something like this will reproduce the error:

```
$BADREF = 42;
process $BADREF 1,2,3;
$BADREF->process(1,2,3);
```

Can't coerce array into hash
: (F) You used an array where a hash was expected, but the array has no information on how to map from keys to array indices. You can do that only with arrays that have a hash reference at index 0.

Can't goto subroutine from an eval-string
: (F) The "goto subroutine" call can't be used to jump out of an eval "string". (You can use it to jump out of an eval {BLOCK}, but you probably don't want to.)

Can't localize pseudo-hash element
: (F) You said something like `local $ar->{'key'}`, where `$ar` is a reference to a pseudo-hash. That hasn't been implemented yet, but you can get a similar effect by localizing the corresponding array element directly -- `local $ar->[$ar->[0]{'key'}]`.

Can't use %%! because Errno.pm is not available
: (F) The first time the `%!` hash is used, perl automatically loads the Errno.pm module. The Errno module is expected to tie the `%!` hash to provide symbolic names for `$!` errno values.

Cannot find an opnumber for "%s"
: (F) A string of a form `CORE::word` was given to *prototype()*, but there is no builtin with the name `word`.

Character class syntax [. .] is reserved for future extensions
: (W) Within regular expression character classes ([]) the syntax beginning with "[." and ending with ".]" is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: "\[." and ".\]".

Character class syntax [: :] is reserved for future extensions
: (W) Within regular expression character classes ([]) the syntax beginning with "[:" and ending with ":]" is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: "\[:" and ":\]".

Character class syntax [= =] is reserved for future extensions
: (W) Within regular expression character classes ([]) the syntax beginning with "[=" and ending with "=]" is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: "\[=" and "=\]".

%s: Eval-group in insecure regular expression
: (F) Perl detected tainted data when trying to compile a regular expression that contains the `(?{ ... })` zero-width assertion, which is unsafe. See the section on *(?{ code })* in the *perlre* manpage, and the *perlsec* manpage.

%s: Eval-group not allowed, use re 'eval'
: (F) A regular expression contained the `(?{ ... })` zero-width assertion, but that construct is only allowed when the `use re 'eval'` pragma is in effect. See the section on *(?{ code })* in the *perlre* manpage.

%s: Eval-group not allowed at run time
: (F) Perl tried to compile a regular expression containing the `(?{ ... })` zero-width assertion at run time, as it would when the pattern contains interpolated values. Since that is a security risk, it is not allowed. If you insist, you may still do this by explicitly building the pattern from an interpolated string at run time and using that in an *eval()*. See the section on *(?{ code })* in the *perlre* manpage.

Explicit blessing to '' (assuming package main)
: (W) You are blessing a reference to a zero length string. This has the effect of blessing the reference into the package main. This is usually not what you want. Consider providing a default target package, e.g. *bless*($ref, `$p` or 'MyPackage');

Illegal hex digit ignored
: (W) You may have tried to use a character other than 0 – 9 or A – F in a hexadecimal number. Interpretation of the hexadecimal number stopped before the illegal character.

No such array field

(F) You tried to access an array as a hash, but the field name used is not defined. The hash at index 0 should map all valid field names to array indices for that to work.

No such field "%s" in variable %s of type %s

(F) You tried to access a field of a typed variable where the type does not know about the field name. The field names are looked up in the `%FIELDS` hash in the type package at compile time. The `%FIELDS` hash is usually set up with the 'fields' pragma.

Out of memory during ridiculously large request

(F) You can't allocate more than 2^31+"small amount" bytes. This error is most likely to be caused by a typo in the Perl program. e.g., `$arr[time]` instead of `$arr[$time]`.

Range iterator outside integer range

(F) One (or both) of the numeric arguments to the range operator ".." are outside the range which can be represented by integers internally. One possible workaround is to force Perl to use magical string increment by prepending "0" to your numbers.

Recursive inheritance detected while looking for method '%s' in package '%s'

(F) More than 100 levels of inheritance were encountered while invoking a method. Probably indicates an unintended loop in your inheritance hierarchy.

Reference found where even-sized list expected

(W) You gave a single reference where Perl was expecting a list with an even number of elements (for assignment to a hash). This usually means that you used the anon hash constructor when you meant to use parens. In any case, a hash requires key/value **pairs**.

```
%hash = { one => 1, two => 2, };   # WRONG
%hash = [ qw/ an anon array / ];   # WRONG
%hash = ( one => 1, two => 2, );   # right
%hash = qw( one 1 two 2 );                 # also fine
```

Undefined value assigned to typeglob

(W) An undefined value was assigned to a typeglob, a la `*foo = undef`. This does nothing. It's possible that you really mean `undef *foo`.

Use of reserved word "%s" is deprecated

(D) The indicated bareword is a reserved word. Future versions of perl may use it as a keyword, so you're better off either explicitly quoting the word in a manner appropriate for its context of use, or using a different name altogether. The warning can be suppressed for subroutine names by either adding a `&` prefix, or using a package qualifier, e.g. `&our()`, or `Foo::our()`.

perl: warning: Setting locale failed.

(S) The whole warning message will look something like:

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
        LC_ALL = "En_US",
        LANG = (unset)
    are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

Exactly what were the failed locale settings varies. In the above the settings were that the LC_ALL was "En_US" and the LANG had no value. This error means that Perl detected that you and/or your system administrator have set up the so-called variable system but Perl could not use those settings. This was not dead serious, fortunately: there is a "default locale" called "C" that Perl can and will use, the script will be run. Before you really fix the problem, however, you will get the same error message each time you run Perl. How to really fix the problem can be found in the *perllocale* manpage section **LOCALE PROBLEMS**.

**Obsolete Diagnostics**

Can't mktemp()

(F) The *mktemp()* routine failed for some reason while trying to process a **–e** switch. Maybe your /tmp partition is full, or clobbered.

Can't write to temp file for **–e**: %s

(F) The write routine failed for some reason while trying to process a **–e** switch. Maybe your /tmp partition is full, or clobbered.

Cannot open temporary file

(F) The create routine failed for some reason while trying to process a **–e** switch. Maybe your /tmp partition is full, or clobbered.

**BUGS**

If you find what you think is a bug, you might check the headers of recently posted articles in the comp.lang.perl.misc newsgroup. There may also be information at http://www.perl.com/perl/, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Make sure you trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to <*perlbug@perl.com*> to be analysed by the Perl porting team.

**SEE ALSO**

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

**HISTORY**

Written by Gurusamy Sarathy <*gsar@umich.edu*>, with many contributions from The Perl Porters.

Send omissions or corrections to <*perlbug@perl.com*>.

## NAME

perldelta – what's new for perl5.004

## DESCRIPTION

This document describes differences between the 5.003 release (as documented in *Programming Perl*, second edition— the Camel Book) and this one.

### Supported Environments

Perl5.004 builds out of the box on Unix, Plan 9, LynxOS, VMS, OS/2, QNX, AmigaOS, and Windows NT. Perl runs on Windows 95 as well, but it cannot be built there, for lack of a reasonable command interpreter.

### Core Changes

Most importantly, many bugs were fixed, including several security problems. See the *Changes* file in the distribution for details.

#### List assignment to `%ENV` works

`%ENV = ()` and `%ENV = @list` now work as expected (except on VMS where it generates a fatal error).

#### "Can't locate Foo.pm in `@INC`" error now lists `@INC`

#### Compilation option: Binary compatibility with 5.003

There is a new Configure question that asks if you want to maintain binary compatibility with Perl 5.003. If you choose binary compatibility, you do not have to recompile your extensions, but you might have symbol conflicts if you embed Perl in another application, just as in the 5.003 release. By default, binary compatibility is preserved at the expense of symbol table pollution.

#### `$PERL5OPT` environment variable

You may now put Perl options in the `$PERL5OPT` environment variable. Unless Perl is running with taint checks, it will interpret this variable as if its contents had appeared on a "#!perl" line at the beginning of your script, except that hyphens are optional. PERL5OPT may only be used to set the following switches: **–[DIMUdmw]**.

#### Limitations on –M, –m, and –T options

The `-M` and `-m` options are no longer allowed on the `#!` line of a script. If a script needs a module, it should invoke it with the `use` pragma.

The **–T** option is also forbidden on the `#!` line of a script, unless it was present on the Perl command line. Due to the way `#!` works, this usually means that **–T** must be in the first argument. Thus:

```
#!/usr/bin/perl -T -w
```

will probably work for an executable script invoked as `scriptname`, while:

```
#!/usr/bin/perl -w -T
```

will probably fail under the same conditions. (Non-Unix systems will probably not follow this rule.) But `perl scriptname` is guaranteed to fail, since then there is no chance of **–T** being found on the command line before it is found on the `#!` line.

#### More precise warnings

If you removed the **–w** option from your Perl 5.003 scripts because it made Perl too verbose, we recommend that you try putting it back when you upgrade to Perl 5.004. Each new perl version tends to remove some undesirable warnings, while adding new warnings that may catch bugs in your scripts.

**Deprecated: Inherited** `AUTOLOAD` for non-methods

Before Perl 5.004, `AUTOLOAD` functions were looked up as methods (using the `@ISA` hierarchy), even when the function to be autoloaded was called as a plain function (e.g. `Foo::bar()`), not a method (e.g. `Foo->bar()` or `$obj->bar()`).

Perl 5.005 will use method lookup only for methods' `AUTOLOAD`s. However, there is a significant base of existing code that may be using the old behavior. So, as an interim step, Perl 5.004 issues an optional warning when a non-method uses an inherited `AUTOLOAD`.

The simple rule is: Inheritance will not work when autoloading non-methods. The simple fix for old code is: In any module that used to depend on inheriting `AUTOLOAD` for non-methods from a base class named `BaseClass`, execute `*AUTOLOAD = \&BaseClass::AUTOLOAD` during startup.

**Previously deprecated** `%OVERLOAD` is no longer usable

Using `%OVERLOAD` to define overloading was deprecated in 5.003. Overloading is now defined using the overload pragma. `%OVERLOAD` is still used internally but should not be used by Perl scripts. See the *overload* manpage for more details.

**Subroutine arguments created only when they're modified**

In Perl 5.004, nonexistent array and hash elements used as subroutine parameters are brought into existence only if they are actually assigned to (via `@_`).

Earlier versions of Perl vary in their handling of such arguments. Perl versions 5.002 and 5.003 always brought them into existence. Perl versions 5.000 and 5.001 brought them into existence only if they were not the first argument (which was almost certainly a bug). Earlier versions of Perl never brought them into existence.

For example, given this code:

```
undef @a; undef %a;
sub show { print $_[0] };
sub change { $_[0]++ };
show($a[2]);
change($a{b});
```

After this code executes in Perl 5.004, `$a{b}` exists but `$a`[2] does not. In Perl 5.002 and 5.003, both `$a{b}` and `$a`[2] would have existed (but `$a`[2]'s value would have been undefined).

**Group vector changeable with** `$)`

The `$)` special variable has always (well, in Perl 5, at least) reflected not only the current effective group, but also the group list as returned by the `getgroups()` C function (if there is one). However, until this release, there has not been a way to call the `setgroups()` C function from Perl.

In Perl 5.004, assigning to `$)` is exactly symmetrical with examining it: The first number in its string value is used as the effective gid; if there are any numbers after the first one, they are passed to the `setgroups()` C function (if there is one).

**Fixed parsing of $$<digit>, &$<digit>, etc.**

Perl versions before 5.004 misinterpreted any type marker followed by "$" and a digit. For example, "$$0" was incorrectly taken to mean "${$}0" instead of "${$0}". This bug is (mostly) fixed in Perl 5.004.

However, the developers of Perl 5.004 could not fix this bug completely, because at least two widely-used modules depend on the old meaning of "$$0" in a string. So Perl 5.004 still interprets "$$<digit>" in the old (broken) way inside strings; but it generates this message as a warning. And in Perl 5.005, this special treatment will cease.

### Fixed localization of $<digit>, $&, etc.

Perl versions before 5.004 did not always properly localize the regex-related special variables. Perl 5.004 does localize them, as the documentation has always said it should. This may result in `$1`, `$2`, etc. no longer being set where existing programs use them.

### No resetting of $. on implicit close

The documentation for Perl 5.0 has always stated that `$.` is *not* reset when an already-open file handle is reopened with no intervening call to `close`. Due to a bug, perl versions 5.000 through 5.003 *did* reset `$.` under that circumstance; Perl 5.004 does not.

`wantarray` may return undef

The `wantarray` operator returns true if a subroutine is expected to return a list, and false otherwise. In Perl 5.004, `wantarray` can also return the undefined value if a subroutine's return value will not be used at all, which allows subroutines to avoid a time-consuming calculation of a return value if it isn't going to be used.

`eval EXPR` determines value of EXPR in scalar context

Perl (version 5) used to determine the value of EXPR inconsistently, sometimes incorrectly using the surrounding context for the determination. Now, the value of EXPR (before being parsed by eval) is always determined in a scalar context. Once parsed, it is executed as before, by providing the context that the scope surrounding the eval provided. This change makes the behavior Perl4 compatible, besides fixing bugs resulting from the inconsistent behavior. This program:

```
@a = qw(time now is time);
print eval @a;
print '|', scalar eval @a;
```

used to print something like "timenowis881399109|4", but now (and in perl4) prints "4|4".

### Changes to tainting checks

A bug in previous versions may have failed to detect some insecure conditions when taint checks are turned on. (Taint checks are used in setuid or setgid scripts, or when explicitly turned on with the `-T` invocation option.) Although it's unlikely, this may cause a previously-working script to now fail -- which should be construed as a blessing, since that indicates a potentially-serious security hole was just plugged.

The new restrictions when tainting include:

No glob() or <*>
: These operators may spawn the C shell (csh), which cannot be made safe. This restriction will be lifted in a future version of Perl when globbing is implemented without the use of an external program.

No spawning if tainted $CDPATH, $ENV, $BASH_ENV
: These environment variables may alter the behavior of spawned programs (especially shells) in ways that subvert security. So now they are treated as dangerous, in the manner of `$IFS` and `$PATH`.

No spawning if tainted $TERM doesn't look like a terminal name
: Some termcap libraries do unsafe things with `$TERM`. However, it would be unnecessarily harsh to treat all `$TERM` values as unsafe, since only shell metacharacters can cause trouble in `$TERM`. So a tainted `$TERM` is considered to be safe if it contains only alphanumerics, underscores, dashes, and colons, and unsafe if it contains other characters (including whitespace).

### New Opcode module and revised Safe module

A new Opcode module supports the creation, manipulation and application of opcode masks. The revised Safe module has a new API and is implemented using the new Opcode module. Please read the new Opcode and Safe documentation.

### Embedding improvements

In older versions of Perl it was not possible to create more than one Perl interpreter instance inside a single process without leaking like a sieve and/or crashing. The bugs that caused this behavior have all been fixed. However, you still must take care when embedding Perl in a C program. See the updated perlembed manpage for tips on how to manage your interpreters.

### Internal change: FileHandle class based on IO::* classes

File handles are now stored internally as type IO::Handle. The FileHandle module is still supported for backwards compatibility, but it is now merely a front end to the IO::* modules -- specifically, IO::Handle, IO::Seekable, and IO::File. We suggest, but do not require, that you use the IO::* modules in new code.

In harmony with this change, `*GLOB{FILEHANDLE}` is now just a backward-compatible synonym for `*GLOB{IO}`.

### Internal change: PerlIO abstraction interface

It is now possible to build Perl with AT&T's sfio IO package instead of stdio. See the *perlapio* manpage for more details, and the *INSTALL* file for how to use it.

### New and changed syntax

$coderef->(PARAMS)

A subroutine reference may now be suffixed with an arrow and a (possibly empty) parameter list. This syntax denotes a call of the referenced subroutine, with the given parameters (if any).

This new syntax follows the pattern of `$hashref->{FOO}` and `$aryref->[$foo]`: You may now write `&$subref($foo)` as `$subref->($foo)`. All of these arrow terms may be chained; thus, `&{$table->{FOO}}($bar)` may now be written `$table->{FOO}->($bar)`.

### New and changed builtin constants

__PACKAGE__

The current package name at compile time, or the undefined value if there is no current package (due to a `package;` directive). Like `__FILE__` and `__LINE__`, `__PACKAGE__` does *not* interpolate into strings.

### New and changed builtin variables

$^E Extended error message on some platforms. (Also known as `$EXTENDED_OS_ERROR` if you `use English`).

$^H The current set of syntax checks enabled by `use strict`. See the documentation of `strict` for more details. Not actually new, but newly documented. Because it is intended for internal use by Perl core components, there is no `use English` long name for this variable.

$^M By default, running out of memory it is not trappable. However, if compiled for this, Perl may use the contents of `$^M` as an emergency pool after *die()*ing with this message. Suppose that your Perl were compiled with −DPERL_EMERGENCY_SBRK and used Perl's malloc. Then

```
$^M = 'a' x (1<<16);
```

would allocate a 64K buffer for use when in emergency. See the *INSTALL* file for information on how to enable this option. As a disincentive to casual use of this advanced feature, there is no `use`

`English` long name for this variable.

**New and changed builtin functions**

delete on slices
: This now works. (e.g. `delete @ENV{'PATH', 'MANPATH'}`)

flock
: is now supported on more platforms, prefers fcntl to lockf when emulating, and always flushes before (un)locking.

printf and sprintf
: Perl now implements these functions itself; it doesn't use the C library function *sprintf()* any more, except for floating-point numbers, and even then only known flags are allowed. As a result, it is now possible to know which conversions and flags will work, and what they will do.

The new conversions in Perl's *sprintf()* are:

```
%i   a synonym for %d
%p   a pointer (the address of the Perl value, in hexadecimal)
%n   special: *stores* the number of characters output so far
     into the next variable in the parameter list
```

The new flags that go between the `%` and the conversion are:

```
#    prefix octal with "0", hex with "0x"
h    interpret integer as C type "short" or "unsigned short"
V    interpret integer as Perl's standard integer type
```

Also, where a number would appear in the flags, an asterisk ("*") may be used instead, in which case Perl uses the next item in the parameter list as the given number (that is, as the field width or precision). If a field width obtained through "*" is negative, it has the same effect as the '−' flag: left-justification.

See the `sprintf` entry in the *perlfunc* manpage for a complete list of conversion and flags.

keys as an lvalue
: As an lvalue, `keys` allows you to increase the number of hash buckets allocated for the given hash. This can gain you a measure of efficiency if you know the hash is going to get big. (This is similar to pre-extending an array by assigning a larger number to $#array.) If you say

```
keys %hash = 200;
```

then `%hash` will have at least 200 buckets allocated for it. These buckets will be retained even if you do `%hash = ()`; use `undef %hash` if you want to free the storage while `%hash` is still in scope. You can't shrink the number of buckets allocated for the hash using `keys` in this way (but you needn't worry about doing this by accident, as trying has no effect).

my() in Control Structures
: You can now use *my()* (with or without the parentheses) in the control expressions of control structures such as:

```
while (defined(my $line = <>)) {
    $line = lc $line;
} continue {
    print $line;
}
```

```
if ((my $answer = <STDIN>) =~ /^y(es)?$/i) {
    user_agrees();
} elsif ($answer =~ /^n(o)?$/i) {
    user_disagrees();
} else {
    chomp $answer;
    die "'$answer' is neither 'yes' nor 'no'";
}
```

Also, you can declare a foreach loop control variable as lexical by preceding it with the word "my". For example, in:

```
foreach my $i (1, 2, 3) {
    some_function();
}
```

$i is a lexical variable, and the scope of `$i` extends to the end of the loop, but not beyond it.

Note that you still cannot use *my()* on global punctuation variables such as `$_` and the like.

pack() and unpack()
: A new format 'w' represents a BER compressed integer (as defined in ASN.1). Its format is a sequence of one or more bytes, each of which provides seven bits of the total value, with the most significant first. Bit eight of each byte is set, except for the last byte, in which bit eight is clear.

  If 'p' or 'P' are given undef as values, they now generate a NULL pointer.

  Both *pack()* and *unpack()* now fail when their templates contain invalid types. (Invalid types used to be ignored.)

sysseek()
: The new *sysseek()* operator is a variant of *seek()* that sets and gets the file's system read/write position, using the *lseek*(2) system call. It is the only reliable way to seek before using *sysread()* or *syswrite()*. Its return value is the new position, or the undefined value on failure.

use VERSION
: If the first argument to `use` is a number, it is treated as a version number instead of a module name. If the version of the Perl interpreter is less than VERSION, then an error message is printed and Perl exits immediately. Because `use` occurs at compile time, this check happens immediately during the compilation process, unlike `require VERSION`, which waits until runtime for the check. This is often useful if you need to check the current Perl version before `use`ing library modules which have changed in incompatible ways from older versions of Perl. (We try not to do this more than we have to.)

use Module VERSION LIST
: If the VERSION argument is present between Module and LIST, then the `use` will call the VERSION method in class Module with the given version as an argument. The default VERSION method, inherited from the UNIVERSAL class, croaks if the given version is larger than the value of the variable `$Module::VERSION`. (Note that there is not a comma after VERSION!)

  This version-checking mechanism is similar to the one currently used in the Exporter module, but it is faster and can be used with modules that don't use the Exporter. It is the recommended method for new code.

prototype(FUNCTION)
: Returns the prototype of a function as a string (or `undef` if the function has no prototype). FUNCTION is a reference to or the name of the function whose prototype you want to retrieve. (Not actually new; just never documented before.)

srand

The default seed for `srand`, which used to be `time`, has been changed. Now it's a heady mix of difficult-to-predict system-dependent values, which should be sufficient for most everyday purposes.

Previous to version 5.004, calling `rand` without first calling `srand` would yield the same sequence of random numbers on most or all machines. Now, when perl sees that you're calling `rand` and haven't yet called `srand`, it calls `srand` with the default seed. You should still call `srand` manually if your code might ever be run on a pre-5.004 system, of course, or if you want a seed other than the default.

$_ as Default

Functions documented in the Camel to default to `$_` now in fact do, and all those that do are so documented in the *perlfunc* manpage.

`m//gc` does not reset search position on failure

The `m//g` match iteration construct has always reset its target string's search position (which is visible through the `pos` operator) when a match fails; as a result, the next `m//g` match after a failure starts again at the beginning of the string. With Perl 5.004, this reset may be disabled by adding the "c" (for "continue") modifier, i.e. `m//gc`. This feature, in conjunction with the `\G` zero-width assertion, makes it possible to chain matches together. See the *perlop* manpage and the *perlre* manpage.

`m//x` ignores whitespace before ?*+{}

The `m//x` construct has always been intended to ignore all unescaped whitespace. However, before Perl 5.004, whitespace had the effect of escaping repeat modifiers like "*" or "?"; for example, `/a *b/x` was (mis)interpreted as `/a\*b/x`. This bug has been fixed in 5.004.

nested `sub{}` closures work now

Prior to the 5.004 release, nested anonymous functions didn't work right. They do now.

formats work right on changing lexicals

Just like anonymous functions that contain lexical variables that change (like a lexical index variable for a `foreach` loop), formats now work properly. For example, this silently failed before (printed only zeros), but is fine now:

```
my $i;
foreach $i ( 1 .. 10 ) {
    write;
}
format =
    my i is @#
    $i
.
```

However, it still fails (without a warning) if the foreach is within a subroutine:

```
my $i;
sub foo {
  foreach $i ( 1 .. 10 ) {
    write;
  }
}
foo;
format =
    my i is @#
    $i
.
```

### New builtin methods

The `UNIVERSAL` package automatically contains the following methods that are inherited by all other classes:

isa(CLASS)

`isa` returns *true* if its object is blessed into a subclass of `CLASS`

`isa` is also exportable and can be called as a sub with two arguments. This allows the ability to check what a reference points to. Example:

```
use UNIVERSAL qw(isa);

if(isa($ref, 'ARRAY')) {
   ...
}
```

can(METHOD)

`can` checks to see if its object has a method called `METHOD`, if it does then a reference to the sub is returned; if it does not then *undef* is returned.

VERSION( [NEED] )

`VERSION` returns the version number of the class (package). If the NEED argument is given then it will check that the current version (as defined by the `$VERSION` variable in the given package) not less than NEED; it will die if this is not the case. This method is normally called as a class method. This method is called automatically by the `VERSION` form of `use`.

```
use A 1.2 qw(some imported subs);
# implies:
A->VERSION(1.2);
```

**NOTE:** `can` directly uses Perl's internal code for method lookup, and `isa` uses a very similar method and caching strategy. This may cause strange effects if the Perl code dynamically changes `@ISA` in any package.

You may add other methods to the UNIVERSAL class via Perl or XS code. You do not need to `use UNIVERSAL` in order to make these methods available to your program. This is necessary only if you wish to have `isa` available as a plain subroutine in the current package.

### TIEHANDLE now supported

See the *perltie* manpage for other kinds of *tie()*s.

TIEHANDLE classname, LIST

This is the constructor for the class. That means it is expected to return an object of some sort. The reference can be used to hold some internal information.

```
sub TIEHANDLE {
    print "<shout>\n";
    my $i;
    return bless \$i, shift;
}
```

PRINT this, LIST

This method will be triggered every time the tied handle is printed to. Beyond its self reference it also expects the list that was passed to the print function.

```
sub PRINT {
    $r = shift;
    $$r++;
    return print join( $, => map {uc} @_), $\;
}
```

PRINTF this, LIST
: This method will be triggered every time the tied handle is printed to with the `printf()` function. Beyond its self reference it also expects the format and list that was passed to the printf function.

```
sub PRINTF {
    shift;
      my $fmt = shift;
    print sprintf($fmt, @_)."\n";
}
```

READ this LIST
: This method will be called when the handle is read from via the `read` or `sysread` functions.

```
sub READ {
    $r = shift;
    my($buf,$len,$offset) = @_;
    print "READ called, \$buf=$buf, \$len=$len, \$offset=$offset";
}
```

READLINE this
: This method will be called when the handle is read from. The method should return undef when there is no more data.

```
sub READLINE {
    $r = shift;
    return "PRINT called $$r times\n"
}
```

GETC this
: This method will be called when the `getc` function is called.

```
sub GETC { print "Don't GETC, Get Perl"; return "a"; }
```

DESTROY this
: As with the other types of ties, this method will be called when the tied handle is about to be destroyed. This is useful for debugging and possibly for cleaning up.

```
sub DESTROY {
    print "</shout>\n";
}
```

### Malloc enhancements

If perl is compiled with the malloc included with the perl distribution (that is, if `perl -V:d_mymalloc` is 'define') then you can print memory statistics at runtime by running Perl thusly:

```
env PERL_DEBUG_MSTATS=2 perl your_script_here
```

The value of 2 means to print statistics after compilation and on exit; with a value of 1, the statistics are printed only on exit. (If you want the statistics at an arbitrary time, you'll need to install the optional module Devel::Peek.)

Three new compilation flags are recognized by malloc.c. (They have no effect if perl is compiled with system *malloc()*.)

-DPERL_EMERGENCY_SBRK
: If this macro is defined, running out of memory need not be a fatal error: a memory pool can allocated by assigning to the special variable `$^M`. See the section on *$^M*.

-DPACK_MALLOC
: Perl memory allocation is by bucket with sizes close to powers of two. Because of these malloc overhead may be big, especially for data of size exactly a power of two. If `PACK_MALLOC` is defined, perl uses a slightly different algorithm for small allocations (up to 64 bytes long), which makes it possible to have overhead down to 1 byte for allocations which are powers of two (and appear quite often).

  Expected memory savings (with 8-byte alignment in `alignbytes`) is about 20% for typical Perl usage. Expected slowdown due to additional malloc overhead is in fractions of a percent (hard to measure, because of the effect of saved memory on speed).

-DTWO_POT_OPTIMIZE
: Similarly to `PACK_MALLOC`, this macro improves allocations of data with size close to a power of two; but this works for big allocations (starting with 16K by default). Such allocations are typical for big hashes and special-purpose scripts, especially image processing.

  On recent systems, the fact that perl requires 2M from system for 1M allocation will not affect speed of execution, since the tail of such a chunk is not going to be touched (and thus will not require real memory). However, it may result in a premature out-of-memory error. So if you will be manipulating very large blocks with sizes close to powers of two, it would be wise to define this macro.

  Expected saving of memory is 0-100% (100% in applications which require most memory in such 2**n chunks); expected slowdown is negligible.

### Miscellaneous efficiency enhancements

Functions that have an empty prototype and that do nothing but return a fixed value are now inlined (e.g. `sub PI () { 3.14159 }`).

Each unique hash key is only allocated once, no matter how many hashes have an entry with that key. So even if you have 100 copies of the same hash, the hash keys never have to be reallocated.

## Support for More Operating Systems

Support for the following operating systems is new in Perl 5.004.

### Win32

Perl 5.004 now includes support for building a "native" perl under Windows NT, using the Microsoft Visual C++ compiler (versions 2.0 and above) or the Borland C++ compiler (versions 5.02 and above). The resulting perl can be used under Windows 95 (if it is installed in the same directory locations as it got installed in Windows NT). This port includes support for perl extension building tools like the *MakeMaker* manpage and the *h2xs* manpage, so that many extensions available on the Comprehensive Perl Archive Network (CPAN) can now be readily built under Windows NT. See http://www.perl.com/ for more information on CPAN and *README.win32* in the perl distribution for more details on how to get started with building this port.

There is also support for building perl under the Cygwin32 environment. Cygwin32 is a set of GNU tools that make it possible to compile and run many UNIX programs under Windows NT by providing a mostly UNIX−like interface for compilation and execution. See *README.cygwin32* in the perl distribution for more details on this port and how to obtain the Cygwin32 toolkit.

### Plan 9

See *README.plan9* in the perl distribution.

### QNX

See *README.qnx* in the perl distribution.

### AmigaOS

See *README.amigaos* in the perl distribution.

## Pragmata

Six new pragmatic modules exist:

use autouse MODULE => qw(sub1 sub2 sub3)
: Defers `require MODULE` until someone calls one of the specified subroutines (which must be exported by MODULE). This pragma should be used with caution, and only when necessary.

use blib

use blib 'dir'
: Looks for MakeMaker-like *'blib'* directory structure starting in *dir* (or current directory) and working back up to five levels of parent directories.

  Intended for use on command line with **−M** option as a way of testing arbitrary scripts against an uninstalled version of a package.

use constant NAME => VALUE
: Provides a convenient interface for creating compile-time constants, See the section on *Constant Functions* in the *perlsub* manpage.

use locale
: Tells the compiler to enable (or disable) the use of POSIX locales for builtin operations.

  When `use locale` is in effect, the current LC_CTYPE locale is used for regular expressions and case mapping; LC_COLLATE for string ordering; and LC_NUMERIC for numeric formating in printf and sprintf (but **not** in print). LC_NUMERIC is always used in write, since lexical scoping of formats is problematic at best.

  Each `use locale` or `no locale` affects statements to the end of the enclosing BLOCK or, if not inside a BLOCK, to the end of the current file. Locales can be switched and queried with *POSIX::setlocale()*.

  See the *perllocale* manpage for more information.

use ops
: Disable unsafe opcodes, or any named opcodes, when compiling Perl code.

use vmsish
: Enable VMS−specific language features. Currently, there are three VMS−specific features available: 'status', which makes `$?` and `system` return genuine VMS status values instead of emulating POSIX; 'exit', which makes `exit` take a genuine VMS status value instead of assuming that `exit 1` is an error; and 'time', which makes all times relative to the local time zone, in the VMS tradition.

## Modules

### Required Updates

Though Perl 5.004 is compatible with almost all modules that work with Perl 5.003, there are a few exceptions:

```
Module   Required Version for Perl 5.004
------   -------------------------------
Filter   Filter-1.12
LWP      libwww-perl-5.08
Tk       Tk400.202 (-w makes noise)
```

Also, the majordomo mailing list program, version 1.94.1, doesn't work with Perl 5.004 (nor with perl 4), because it executes an invalid regular expression. This bug is fixed in majordomo version 1.94.2.

### Installation directories

The *installperl* script now places the Perl source files for extensions in the architecture-specific library directory, which is where the shared libraries for extensions have always been. This change is intended to allow administrators to keep the Perl 5.004 library directory unchanged from a previous version, without running the risk of binary incompatibility between extensions' Perl source and shared libraries.

### Module information summary

Brand new modules, arranged by topic rather than strictly alphabetically:

```
CGI.pm                Web server interface ("Common Gateway Interface")
CGI/Apache.pm         Support for Apache's Perl module
CGI/Carp.pm           Log server errors with helpful context
CGI/Fast.pm           Support for FastCGI (persistent server process)
CGI/Push.pm           Support for server push
CGI/Switch.pm         Simple interface for multiple server types

CPAN                  Interface to Comprehensive Perl Archive Network
CPAN::FirstTime       Utility for creating CPAN configuration file
CPAN::Nox             Runs CPAN while avoiding compiled extensions

IO.pm                 Top-level interface to IO::* classes
IO/File.pm            IO::File extension Perl module
IO/Handle.pm          IO::Handle extension Perl module
IO/Pipe.pm            IO::Pipe extension Perl module
IO/Seekable.pm        IO::Seekable extension Perl module
IO/Select.pm          IO::Select extension Perl module
IO/Socket.pm          IO::Socket extension Perl module

Opcode.pm             Disable named opcodes when compiling Perl code

ExtUtils/Embed.pm     Utilities for embedding Perl in C programs
ExtUtils/testlib.pm   Fixes up @INC to use just-built extension

FindBin.pm            Find path of currently executing program
```

```
Class/Struct.pm      Declare struct-like datatypes as Perl classes
File/stat.pm         By-name interface to Perl's builtin stat
Net/hostent.pm       By-name interface to Perl's builtin gethost*
Net/netent.pm        By-name interface to Perl's builtin getnet*
Net/protoent.pm      By-name interface to Perl's builtin getproto*
Net/servent.pm       By-name interface to Perl's builtin getserv*
Time/gmtime.pm       By-name interface to Perl's builtin gmtime
Time/localtime.pm    By-name interface to Perl's builtin localtime
Time/tm.pm           Internal object for Time::{gm,local}time
User/grent.pm        By-name interface to Perl's builtin getgr*
User/pwent.pm        By-name interface to Perl's builtin getpw*

Tie/RefHash.pm       Base class for tied hashes with references as keys

UNIVERSAL.pm         Base class for *ALL* classes
```

### Fcntl

New constants in the existing Fcntl modules are now supported, provided that your operating system happens to support them:

```
F_GETOWN F_SETOWN
O_ASYNC O_DEFER O_DSYNC O_FSYNC O_SYNC
O_EXLOCK O_SHLOCK
```

These constants are intended for use with the Perl operators *sysopen()* and *fcntl()* and the basic database modules like SDBM_File. For the exact meaning of these and other Fcntl constants please refer to your operating system's documentation for *fcntl()* and *open()*.

In addition, the Fcntl module now provides these constants for use with the Perl operator *flock()*:

```
    LOCK_SH LOCK_EX LOCK_NB LOCK_UN
```

These constants are defined in all environments (because where there is no *flock()* system call, Perl emulates it). However, for historical reasons, these constants are not exported unless they are explicitly requested with the ":flock" tag (e.g. `use Fcntl ':flock'`).

### IO

The IO module provides a simple mechanism to load all of the IO modules at one go. Currently this includes:

```
IO::Handle
IO::Seekable
IO::File
IO::Pipe
IO::Socket
```

For more information on any of these modules, please see its respective documentation.

### Math::Complex

The Math::Complex module has been totally rewritten, and now supports more operations. These are overloaded:

```
+ - * / ** <=> neg ~ abs sqrt exp log sin cos atan2 "" (stringify)
```

And these functions are now exported:

```
pi i Re Im arg
log10 logn ln cbrt root
tan
csc sec cot
asin acos atan
acsc asec acot
sinh cosh tanh
csch sech coth
asinh acosh atanh
acsch asech acoth
cplx cplxe
```

### Math::Trig

This new module provides a simpler interface to parts of Math::Complex for those who need trigonometric functions only for real numbers.

### DB_File

There have been quite a few changes made to DB_File. Here are a few of the highlights:

- Fixed a handful of bugs.
- By public demand, added support for the standard hash function *exists()*.
- Made it compatible with Berkeley DB 1.86.
- Made negative subscripts work with RECNO interface.
- Changed the default flags from O_RDWR to O_CREAT|O_RDWR and the default mode from 0640 to 0666.
- Made DB_File automatically import the *open()* constants (O_RDWR, O_CREAT etc.) from Fcntl, if available.
- Updated documentation.

Refer to the HISTORY section in DB_File.pm for a complete list of changes. Everything after DB_File 1.01 has been added since 5.003.

### Net::Ping

Major rewrite – support added for both udp echo and real icmp pings.

### Object-oriented overrides for builtin operators

Many of the Perl builtins returning lists now have object-oriented overrides. These are:

```
File::stat
Net::hostent
Net::netent
Net::protoent
Net::servent
Time::gmtime
Time::localtime
User::grent
User::pwent
```

For example, you can now say

```
use File::stat;
use User::pwent;
$his = (stat($filename)->st_uid == pwent($whoever)->pw_uid);
```

### Utility Changes

#### pod2html

Sends converted HTML to standard output

The *pod2html* utility included with Perl 5.004 is entirely new. By default, it sends the converted HTML to its standard output, instead of writing it to a file like Perl 5.003's *pod2html* did. Use the **--outfile=FILENAME** option to write to a file.

#### xsubpp

`void` XSUBs now default to returning nothing

Due to a documentation/implementation bug in previous versions of Perl, XSUBs with a return type of `void` have actually been returning one value. Usually that value was the GV for the XSUB, but sometimes it was some already freed or reused value, which would sometimes lead to program failure.

In Perl 5.004, if an XSUB is declared as returning `void`, it actually returns no value, i.e. an empty list (though there is a backward-compatibility exception; see below). If your XSUB really does return an SV, you should give it a return type of `SV *`.

For backward compatibility, *xsubpp* tries to guess whether a `void` XSUB is really `void` or if it wants to return an `SV *`. It does so by examining the text of the XSUB: if *xsubpp* finds what looks like an assignment to `ST(0)`, it assumes that the XSUB's return type is really `SV *`.

### C Language API Changes

`gv_fetchmethod` and `perl_call_sv`

The `gv_fetchmethod` function finds a method for an object, just like in Perl 5.003. The GV it returns may be a method cache entry. However, in Perl 5.004, method cache entries are not visible to users; therefore, they can no longer be passed directly to `perl_call_sv`. Instead, you should use the `GvCV` macro on the GV to extract its CV, and pass the CV to `perl_call_sv`.

The most likely symptom of passing the result of `gv_fetchmethod` to `perl_call_sv` is Perl's producing an ''Undefined subroutine called'' error on the *second* call to a given method (since there is no cache on the first call).

`perl_eval_pv`

A new function handy for eval'ing strings of Perl code inside C code. This function returns the value from the eval statement, which can be used instead of fetching globals from the symbol table. See the *perlguts* manpage, the *perlembed* manpage and the *perlcall* manpage for details and examples.

Extended API for manipulating hashes

Internal handling of hash keys has changed. The old hashtable API is still fully supported, and will likely remain so. The additions to the API allow passing keys as `SV*`s, so that `tied` hashes can be given real scalars as keys rather than plain strings (nontied hashes still can only use strings as keys). New extensions must use the new hash access functions and macros if they wish to use `SV*` keys. These additions also make it feasible to manipulate `HE*`s (hash entries), which can be more efficient. See the *perlguts* manpage for details.

### Documentation Changes

Many of the base and library pods were updated. These new pods are included in section 1:

the *perldelta* manpage

This document.

the *perlfaq* manpage
Frequently asked questions.

the *perllocale* manpage
Locale support (internationalization and localization).

the *perltoot* manpage
Tutorial on Perl OO programming.

the *perlapio* manpage
Perl internal IO abstraction interface.

the *perlmodlib* manpage
Perl module library and recommended practice for module creation. Extracted from the *perlmod* manpage (which is much smaller as a result).

the *perldebug* manpage
Although not new, this has been massively updated.

the *perlsec* manpage
Although not new, this has been massively updated.

**New Diagnostics**

Several new conditions will trigger warnings that were silent before. Some only affect certain platforms. The following new warnings and errors outline these. These messages are classified as follows (listed in increasing order of desperation):

```
(W) A warning (optional).
(D) A deprecation (optional).
(S) A severe warning (mandatory).
(F) A fatal error (trappable).
(P) An internal error you should never see (trappable).
(X) A very fatal error (nontrappable).
(A) An alien error message (not generated by Perl).
```

‘‘my’’ variable %s masks earlier declaration in same scope
(W) A lexical variable has been redeclared in the same scope, effectively eliminating all access to the previous instance. This is almost always a typographical error. Note that the earlier variable will still exist until the end of the scope or until all closure referents to it are destroyed.

%s argument is not a HASH element or slice
(F) The argument to *delete()* must be either a hash element, such as

```
$foo{$bar}
$ref->[12]->{"susie"}
```

or a hash slice, such as

```
@foo{$bar, $baz, $xyzzy}
@{$ref->[12]}{"susie", "queue"}
```

Allocation too large: %lx
(X) You can’t allocate more than 64K on an MS−DOS machine.

Allocation too large
(F) You can’t allocate more than 2^31+"small amount’’ bytes.

Applying %s to %s will act on scalar(%s)
(W) The pattern match (//), substitution (s///), and transliteration (tr///) operators work on scalar values. If you apply one of them to an array or a hash, it will convert the array or hash to a scalar value -- the length of an array, or the population info of a hash -- and then work on that scalar value. This

is probably not what you meant to do. See the `grep` entry in the *perlfunc* manpage and the `map` entry in the *perlfunc* manpage for alternatives.

Attempt to free nonexistent shared string
: (P) Perl maintains a reference counted internal table of strings to optimize the storage and access of hash keys and other strings. This indicates someone tried to decrement the reference count of a string that can no longer be found in the table.

Attempt to use reference as lvalue in substr
: (W) You supplied a reference as the first argument to *substr()* used as an lvalue, which is pretty strange. Perhaps you forgot to dereference it first. See the `substr` entry in the *perlfunc* manpage.

Bareword "%s" refers to nonexistent package
: (W) You used a qualified bareword of the form `Foo::`, but the compiler saw no other uses of that namespace before that point. Perhaps you need to predeclare a package?

Can't redefine active sort subroutine %s
: (F) Perl optimizes the internal handling of sort subroutines and keeps pointers into them. You tried to redefine one such sort subroutine when it was currently active, which is not allowed. If you really want to do this, you should write `sort { &func } @x` instead of `sort func @x`.

Can't use bareword '%s' as %s ref while "strict refs" in use
: (F) Only hard references are allowed by "strict refs". Symbolic references are disallowed. See the *perlref* manpage.

Cannot resolve method '%s' overloading '%s' in package '%s'
: (P) Internal error trying to resolve overloading specified by a method name (as opposed to a subroutine reference).

Constant subroutine %s redefined
: (S) You redefined a subroutine which had previously been eligible for inlining. See the section on *Constant Functions* in the *perlsub* manpage for commentary and workarounds.

Constant subroutine %s undefined
: (S) You undefined a subroutine which had previously been eligible for inlining. See the section on *Constant Functions* in the *perlsub* manpage for commentary and workarounds.

Copy method did not return a reference
: (F) The method which overloads "=" is buggy. See the section on *Copy Constructor* in the *overload* manpage.

Died
: (F) You passed *die()* an empty string (the equivalent of `die ""`) or you called it with no args and both `$@` and `$_` were empty.

Exiting pseudo-block via %s
: (W) You are exiting a rather special block construct (like a sort block or subroutine) by unconventional means, such as a goto, or a loop control statement. See the `sort` entry in the *perlfunc* manpage.

Identifier too long
: (F) Perl limits identifiers (names for variables, functions, etc.) to 252 characters for simple names, somewhat more for compound names (like `$A::B`). You've exceeded Perl's limits. Future versions of Perl are likely to eliminate these arbitrary limitations.

Illegal character %s (carriage return)
: (F) A carriage return character was found in the input. This is an error, and not a warning, because carriage return characters can break multi-line strings, including here documents (e.g., `print <<EOF;`).

Illegal switch in PERL5OPT: %s
: (X) The PERL5OPT environment variable may only be used to set the following switches: **–[DIMUdmw]**.

Integer overflow in hex number
: (S) The literal hex number you have specified is too big for your architecture. On a 32-bit architecture the largest hex literal is 0xFFFFFFFF.

Integer overflow in octal number
: (S) The literal octal number you have specified is too big for your architecture. On a 32-bit architecture the largest octal literal is 037777777777.

internal error: glob failed
: (P) Something went wrong with the external *program* (s) used for `glob` and `<*.c>`. This may mean that your csh (C shell) is broken. If so, you should change all of the csh-related variables in config.sh: If you have tcsh, make the variables refer to it as if it were csh (e.g. `full_csh='/usr/bin/tcsh'`); otherwise, make them all empty (except that `d_csh` should be `'undef'`) so that Perl will think csh is missing. In either case, after editing config.sh, run `./Configure -S` and rebuild Perl.

Invalid conversion in %s: "%s"
: (W) Perl does not understand the given format conversion. See the `sprintf` entry in the *perlfunc* manpage.

Invalid type in pack: '%s'
: (F) The given character is not a valid pack type. See the `pack` entry in the *perlfunc* manpage.

Invalid type in unpack: '%s'
: (F) The given character is not a valid unpack type. See the `unpack` entry in the *perlfunc* manpage.

Name "%s::%s" used only once: possible typo
: (W) Typographical errors often show up as unique variable names. If you had a good reason for having a unique name, then just mention it again somehow to suppress the message (the `use vars` pragma is provided for just this purpose).

Null picture in formline
: (F) The first argument to formline must be a valid format picture specification. It was found to be empty, which probably means you supplied it an uninitialized value. See the *perlform* manpage.

Offset outside string
: (F) You tried to do a read/write/send/recv operation with an offset pointing outside the buffer. This is difficult to imagine. The sole exception to this is that `sysread()` ing past the buffer will extend the buffer and zero pad the new area.

Out of memory!
: (X|F) The *malloc()* function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request.

  The request was judged to be small, so the possibility to trap it depends on the way Perl was compiled. By default it is not trappable. However, if compiled for this, Perl may use the contents of `$^M` as an emergency pool after *die()*ing with this message. In this case the error is trappable *once*.

Out of memory during request for %s
: (F) The *malloc()* function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request. However, the request was judged large enough (compile-time default is 64K), so a possibility to shut down by trapping this error is granted.

panic: frexp
: (P) The library function *frexp()* failed, making *printf*("%f") impossible.

Possible attempt to put comments in qw() list
: (W) *qw()* lists contain items separated by whitespace; as with literal strings, comment characters are not ignored, but are instead treated as literal data. (You may have used different delimiters than the parentheses shown here; braces are also frequently used.)

  You probably wrote something like this:

```
@list = qw(
    a # a comment
    b # another comment
);
```

when you should have written this:

```
@list = qw(
    a
    b
);
```

If you really want comments, build your list the old-fashioned way, with quotes and commas:

```
@list = (
    'a',    # a comment
    'b',    # another comment
);
```

Possible attempt to separate words with commas
: (W) *qw()* lists contain items separated by whitespace; therefore commas aren't needed to separate the items. (You may have used different delimiters than the parentheses shown here; braces are also frequently used.)

You probably wrote something like this:

```
qw! a, b, c !;
```

which puts literal commas into some of the list items. Write it without commas if you don't want them to appear in your data:

```
qw! a b c !;
```

Scalar value @%s{%s} better written as $%s{%s}
: (W) You've used a hash slice (indicated by @) to select a single element of a hash. Generally it's better to ask for a scalar value (indicated by $). The difference is that `$foo{&bar}` always behaves like a scalar, both when assigning to it and when evaluating its argument, while `@foo{&bar}` behaves like a list when you assign to it, and provides a list context to its subscript, which can do weird things if you're expecting only one subscript.

Stub found while resolving method '%s' overloading '%s' in package '%s'
: (P) Overloading resolution over `@ISA` tree may be broken by importing stubs. Stubs should never be implicitely created, but explicit calls to `can` may break this.

Too late for "**−T**" option
: (X) The #! line (or local equivalent) in a Perl script contains the **−T** option, but Perl was not invoked with **−T** in its argument list. This is an error because, by the time Perl discovers a **−T** in a script, it's too late to properly taint everything from the environment. So Perl gives up.

untie attempted while %d inner references still exist
: (W) A copy of the object returned from `tie` (or `tied`) was still valid when `untie` was called.

Unrecognized character %s
: (F) The Perl parser has no idea what to do with the specified character in your Perl script (or eval). Perhaps you tried to run a compressed script, a binary program, or a directory as a Perl program.

Unsupported function fork
: (F) Your version of executable does not support forking.

Note that under some systems, like OS/2, there may be different flavors of Perl executables, some of which may support fork, some not. Try changing the name you call Perl by to `perl_`, `perl__`, and so on.

Use of "$$<digit>" to mean "${$}<digit>" is deprecated

(D) Perl versions before 5.004 misinterpreted any type marker followed by "$" and a digit. For example, "$$0" was incorrectly taken to mean "${$}0" instead of "${$0}". This bug is (mostly) fixed in Perl 5.004.

However, the developers of Perl 5.004 could not fix this bug completely, because at least two widely-used modules depend on the old meaning of "$$0" in a string. So Perl 5.004 still interprets "$$<digit>" in the old (broken) way inside strings; but it generates this message as a warning. And in Perl 5.005, this special treatment will cease.

Value of %s can be "0"; test with defined()

(W) In a conditional expression, you used <HANDLE>, <*> (glob), `each()`, or `readdir()` as a boolean value. Each of these constructs can return a value of "0"; that would make the conditional expression false, which is probably not what you intended. When using these constructs in conditional expressions, test their values with the `defined` operator.

Variable "%s" may be unavailable

(W) An inner (nested) *anonymous* subroutine is inside a *named* subroutine, and outside that is another subroutine; and the anonymous (innermost) subroutine is referencing a lexical variable defined in the outermost subroutine. For example:

```
sub outermost { my $a; sub middle { sub { $a } } }
```

If the anonymous subroutine is called or referenced (directly or indirectly) from the outermost subroutine, it will share the variable as you would expect. But if the anonymous subroutine is called or referenced when the outermost subroutine is not active, it will see the value of the shared variable as it was before and during the *first* call to the outermost subroutine, which is probably not what you want.

In these circumstances, it is usually best to make the middle subroutine anonymous, using the `sub {}` syntax. Perl has specific support for shared variables in nested anonymous subroutines; a named subroutine in between interferes with this feature.

Variable "%s" will not stay shared

(W) An inner (nested) *named* subroutine is referencing a lexical variable defined in an outer subroutine.

When the inner subroutine is called, it will probably see the value of the outer subroutine's variable as it was before and during the *first* call to the outer subroutine; in this case, after the first call to the outer subroutine is complete, the inner and outer subroutines will no longer share a common value for the variable. In other words, the variable will no longer be shared.

Furthermore, if the outer subroutine is anonymous and references a lexical variable outside itself, then the outer and inner subroutines will *never* share the given variable.

This problem can usually be solved by making the inner subroutine anonymous, using the `sub {}` syntax. When inner anonymous subs that reference variables in outer subroutines are called or referenced, they are automatically rebound to the current values of such variables.

Warning: something's wrong

(W) You passed *warn()* an empty string (the equivalent of `warn ""`) or you called it with no args and `$_` was empty.

Ill-formed logical name |%s| in prime_env_iter

(W) A warning peculiar to VMS. A logical name was encountered when preparing to iterate over `%ENV` which violates the syntactic rules governing logical names. Since it cannot be translated normally, it is skipped, and will not appear in `%ENV`. This may be a benign occurrence, as some software packages might directly modify logical name tables and introduce nonstandard names, or it may indicate that a logical name table has been corrupted.

Got an error from DosAllocMem
: (P) An error peculiar to OS/2. Most probably you're using an obsolete version of Perl, and this should not happen anyway.

Malformed PERLLIB_PREFIX
: (F) An error peculiar to OS/2. PERLLIB_PREFIX should be of the form

```
prefix1;prefix2
```

or

```
prefix1 prefix2
```

with nonempty prefix1 and prefix2. If `prefix1` is indeed a prefix of a builtin library search path, prefix2 is substituted. The error may appear if components are not found, or are too long. See "PERLLIB_PREFIX" in *README.os2*.

PERL_SH_DIR too long
: (F) An error peculiar to OS/2. PERL_SH_DIR is the directory to find the `sh`-shell in. See "PERL_SH_DIR" in *README.os2*.

Process terminated by SIG%s
: (W) This is a standard message issued by OS/2 applications, while *nix applications die in silence. It is considered a feature of the OS/2 port. One can easily disable this by appropriate sighandlers, see the section on *Signals* in the *perlipc* manpage. See also "Process terminated by SIGTERM/SIGINT" in *README.os2*.

**BUGS**

If you find what you think is a bug, you might check the headers of recently posted articles in the comp.lang.perl.misc newsgroup. There may also be information at http://www.perl.com/perl/, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Make sure you trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to <*perlbug@perl.com*> to be analysed by the Perl porting team.

**SEE ALSO**

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl. This file has been significantly updated for 5.004, so even veteran users should look through it.

The *README* file for general stuff.

The *Copying* file for copyright information.

**HISTORY**

Constructed by Tom Christiansen, grabbing material with permission from innumerable contributors, with kibitzing by more than a few Perl porters.

Last update: Wed May 14 11:14:09 EDT 1997

**NAME**

perlfaq − frequently asked questions about Perl ($Date: 1998/07/20 23:12:17 $)

**DESCRIPTION**

This document is structured into the following sections:

perlfaq: Structural overview of the FAQ.
This document.

the *perlfaq1* manpage: General Questions About Perl
Very general, high-level information about Perl.

the *perlfaq2* manpage: Obtaining and Learning about Perl
Where to find source and documentation to Perl, support, and related matters.

the *perlfaq3* manpage: Programming Tools
Programmer tools and programming support.

the *perlfaq4* manpage: Data Manipulation
Manipulating numbers, dates, strings, arrays, hashes, and miscellaneous data issues.

the *perlfaq5* manpage: Files and Formats
I/O and the "f" issues: filehandles, flushing, formats and footers.

the *perlfaq6* manpage: Regexps
Pattern matching and regular expressions.

the *perlfaq7* manpage: General Perl Language Issues
General Perl language issues that don't clearly fit into any of the other sections.

the *perlfaq8* manpage: System Interaction
Interprocess communication (IPC), control over the user-interface (keyboard, screen and pointing devices).

the *perlfaq9* manpage: Networking
Networking, the Internet, and a few on the web.

### Where to get this document

This document is posted regularly to comp.lang.perl.announce and several other related newsgroups. It is available in a variety of formats from CPAN in the /CPAN/doc/FAQs/FAQ/ directory, or on the web at http://www.perl.com/perl/faq/ .

### How to contribute to this document

You may mail corrections, additions, and suggestions to perlfaq-suggestions@perl.com . This alias should not be used to *ask* FAQs. It's for fixing the current FAQ.

### What will happen if you mail your Perl programming problems to the authors

Your questions will probably go unread, unless they're suggestions of new questions to add to the FAQ, in which case they should have gone to the perlfaq-suggestions@perl.com instead.

You should have read section 2 of this faq. There you would have learned that comp.lang.perl.misc is the appropriate place to go for free advice. If your question is really important and you require a prompt and correct answer, you should hire a consultant.

**Credits**

When I first began the Perl FAQ in the late 80s, I never realized it would have grown to over a hundred pages, nor that Perl would ever become so popular and widespread. This document could not have been written without the tremendous help provided by Larry Wall and the rest of the Perl Porters.

**Author and Copyright Information**

### Bundled Distributions

When included as part of the Standard Version of Perl, or as part of its complete documentation whether printed or otherwise, this work may be distributed only under the terms of Perl's Artistic License. Any distribution of this file or derivatives thereof *outside* of that package require that special arrangements be made with copyright holder.

Irrespective of its distribution, all code examples in these files are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

### Disclaimer

This information is offered in good faith and in the hope that it may be of use, but is not guaranteed to be correct, up to date, or suitable for any particular purpose whatsoever. The authors accept no liability in respect of this information or its use.

**Changes**

22/June/98
Significant changes throughout in preparation for the 5.005 release.

24/April/97
Style and whitespace changes from Chip, new question on reading one character at a time from a terminal using POSIX from Tom.

23/April/97
Added http://www.oasis.leo.org/perl/ to the *perlfaq2* manpage. Style fix to the *perlfaq3* manpage. Added floating point precision, fixed complex number arithmetic, cross-references, caveat for Text::Wrap, alternative answer for initial capitalizing, fixed incorrect regexp, added example of Tie::IxHash to the *perlfaq4* manpage. Added example of passing and storing filehandles, added commify to the *perlfaq5* manpage. Restored variable suicide, and added mass commenting to the *perlfaq7* manpage. Added Net::Telnet, fixed backticks, added reader/writer pair to telnet question, added FindBin, grouped module questions together in the *perlfaq8* manpage. Expanded caveats for the simple URL extractor, gave LWP example, added CGI security question, expanded on the mail address answer in the *perlfaq9* manpage.

25/March/97
Added more info to the binary distribution section of the *perlfaq2* manpage. Added Net::Telnet to the *perlfaq6* manpage. Fixed typos in the *perlfaq8* manpage. Added mail sending example to the *perlfaq9* manpage. Added Merlyn's columns to the *perlfaq2* manpage.

18/March/97
Added the DATE to the NAME section, indicating which sections have changed.

Mentioned SIGPIPE and the *perlipc* manpage in the forking open answer in the *perlfaq8* manpage.

Fixed description of a regular expression in the *perlfaq4* manpage.

17/March/97 Version
Various typos fixed throughout.

Added new question on Perl BNF on the *perlfaq7* manpage.

Initial Release: 11/March/97
This is the initial release of version 3 of the FAQ; consequently there have been no changes since its initial release.

**NAME**

perlfaq1 − General Questions About Perl ($Revision: 1.14 $, `$Date:` 1998/06/14 22:15:25 $)

**DESCRIPTION**

This section of the FAQ answers very general, high-level questions about Perl.

### What is Perl?

Perl is a high-level programming language with an eclectic heritage written by Larry Wall and a cast of thousands. It derives from the ubiquitous C programming language and to a lesser extent from sed, awk, the Unix shell, and at least a dozen other tools and languages. Perl's process, file, and text manipulation facilities make it particularly well-suited for tasks involving quick prototyping, system utilities, software tools, system management tasks, database access, graphical programming, networking, and world wide web programming. These strengths make it especially popular with system administrators and CGI script authors, but mathematicians, geneticists, journalists, and even managers also use Perl. Maybe you should, too.

### Who supports Perl? Who develops it? Why is it free?

The original culture of the pre-populist Internet and the deeply-held beliefs of Perl's author, Larry Wall, gave rise to the free and open distribution policy of perl. Perl is supported by its users. The core, the standard Perl library, the optional modules, and the documentation you're reading now were all written by volunteers. See the personal note at the end of the README file in the perl source distribution for more details.

In particular, the core development team (known as the Perl Porters) are a rag-tag band of highly altruistic individuals committed to producing better software for free than you could hope to purchase for money. You may snoop on pending developments via news://genetics.upenn.edu/perl.porters-gw/ and http://www.frii.com/~gnat/perl/porters/summary.html.

While the GNU project includes Perl in its distributions, there's no such thing as "GNU Perl". Perl is not produced nor maintained by the Free Software Foundation. Perl's licensing terms are also more open than GNU software's tend to be.

You can get commercial support of Perl if you wish, although for most users the informal support will more than suffice. See the answer to "Where can I buy a commercial version of perl?" for more information.

### Which version of Perl should I use?

You should definitely use version 5. Version 4 is old, limited, and no longer maintained; its last patch (4.036) was in 1992. The most recent production release is 5.004_01. Further references to the Perl language in this document refer to this production release unless otherwise specified. There may be one or more official bug fixes for 5.004_01 by the time you read this, and also perhaps some experimental versions on the way to the next release.

### What are perl4 and perl5?

Perl4 and perl5 are informal names for different versions of the Perl programming language. It's easier to say "perl5" than it is to say "the *5*(.004) release of Perl", but some people have interpreted this to mean there's a language called "perl5", which isn't the case. Perl5 is merely the popular name for the fifth major release (October 1994), while perl4 was the fourth major release (March 1991). There was also a perl1 (in January 1988), a perl2 (June 1988), and a perl3 (October 1989).

The 5.0 release is, essentially, a complete rewrite of the perl source code from the ground up. It has been modularized, object-oriented, tweaked, trimmed, and optimized until it almost doesn't look like the old code. However, the interface is mostly the same, and compatibility with previous releases is very high.

To avoid the "what language is perl5?" confusion, some people prefer to simply use "perl" to refer to the latest version of perl and avoid using "perl5" altogether. It's not really that big a deal, though.

See the *perlhist* manpage for a history of Perl revisions.

### How stable is Perl?

Production releases, which incorporate bug fixes and new functionality, are widely tested before release. Since the 5.000 release, we have averaged only about one production release per year.

Larry and the Perl development team occasionally make changes to the internal core of the language, but all possible efforts are made toward backward compatibility. While not quite all perl4 scripts run flawlessly under perl5, an update to perl should nearly never invalidate a program written for an earlier version of perl (barring accidental bug fixes and the rare new keyword).

### Is Perl difficult to learn?

No, Perl is easy to start learning -- and easy to keep learning. It looks like most programming languages you're likely to have experience with, so if you've ever written an C program, an awk script, a shell script, or even BASIC program, you're already part way there.

Most tasks only require a small subset of the Perl language. One of the guiding mottos for Perl development is "there's more than one way to do it" (TMTOWTDI, sometimes pronounced "tim toady"). Perl's learning curve is therefore shallow (easy to learn) and long (there's a whole lot you can do if you really want).

Finally, Perl is (frequently) an interpreted language. This means that you can write your programs and test them without an intermediate compilation step, allowing you to experiment and test/debug quickly and easily. This ease of experimentation flattens the learning curve even more.

Things that make Perl easier to learn: Unix experience, almost any kind of programming experience, an understanding of regular expressions, and the ability to understand other people's code. If there's something you need to do, then it's probably already been done, and a working example is usually available for free. Don't forget the new perl modules, either. They're discussed in Part 3 of this FAQ, along with the CPAN, which is discussed in Part 2.

### How does Perl compare with other languages like Java, Python, REXX, Scheme, or Tcl?

Favorably in some areas, unfavorably in others. Precisely which areas are good and bad is often a personal choice, so asking this question on Usenet runs a strong risk of starting an unproductive Holy War.

Probably the best thing to do is try to write equivalent code to do a set of tasks. These languages have their own newsgroups in which you can learn about (but hopefully not argue about) them.

### Can I do [task] in Perl?

Perl is flexible and extensible enough for you to use on almost any task, from one-line file-processing tasks to complex systems. For many people, Perl serves as a great replacement for shell scripting. For others, it serves as a convenient, high-level replacement for most of what they'd program in low-level languages like C or C++. It's ultimately up to you (and possibly your management ...) which tasks you'll use Perl for and which you won't.

If you have a library that provides an API, you can make any component of it available as just another Perl function or variable using a Perl extension written in C or C++ and dynamically linked into your main perl interpreter. You can also go the other direction, and write your main program in C or C++, and then link in some Perl code on the fly, to create a powerful application.

That said, there will always be small, focused, special-purpose languages dedicated to a specific problem domain that are simply more convenient for certain kinds of problems. Perl tries to be all things to all people, but nothing special to anyone. Examples of specialized languages that come to mind include prolog and matlab.

### When shouldn't I program in Perl?

When your manager forbids it -- but do consider replacing them :−).

Actually, one good reason is when you already have an existing application written in another language that's all done (and done well), or you have an application language specifically designed for a certain task (e.g. prolog, make).

For various reasons, Perl is probably not well-suited for real-time embedded systems, low-level operating systems development work like device drivers or context-switching code, complex multithreaded shared-memory applications, or extremely large applications. You'll notice that perl is not itself written in Perl.

The new native-code compiler for Perl may reduce the limitations given in the previous statement to some degree, but understand that Perl remains fundamentally a dynamically typed language, and not a statically typed one. You certainly won't be chastized if you don't trust nuclear-plant or brain-surgery monitoring code to it. And Larry will sleep easier, too -- Wall Street programs not withstanding. :−)

### What's the difference between "perl" and "Perl"?

One bit. Oh, you weren't talking ASCII? :−) Larry now uses ''Perl'' to signify the language proper and ''perl'' the implementation of it, i.e. the current interpreter. Hence Tom's quip that ''Nothing but perl can parse Perl.'' You may or may not choose to follow this usage. For example, parallelism means ''awk and perl'' and ''Python and Perl'' look ok, while ''awk and Perl'' and ''Python and perl'' do not.

### Is it a Perl program or a Perl script?

It doesn't matter.

In ''standard terminology'' a *program* has been compiled to physical machine code once, and can then be be run multiple times, whereas a *script* must be translated by a program each time it's used. Perl programs, however, are usually neither strictly compiled nor strictly interpreted. They can be compiled to a byte code form (something of a Perl virtual machine) or to completely different languages, like C or assembly language. You can't tell just by looking whether the source is destined for a pure interpreter, a parse-tree interpreter, a byte code interpreter, or a native-code compiler, so it's hard to give a definitive answer here.

### What is a JAPH?

These are the ''just another perl hacker'' signatures that some people sign their postings with. About 100 of the of the earlier ones are available from http://www.perl.com/CPAN/misc/japh .

### Where can I get a list of Larry Wall witticisms?

Over a hundred quips by Larry, from postings of his or source code, can be found at http://www.perl.com/CPAN/misc/lwall-quotes .

### How can I convince my sysadmin/supervisor/employees to use version (5/5.004/Perl instead of some other language)?

If your manager or employees are wary of unsupported software, or software which doesn't officially ship with your Operating System, you might try to appeal to their self-interest. If programmers can be more productive using and utilizing Perl constructs, functionality, simplicity, and power, then the typical manager/supervisor/employee may be persuaded. Regarding using Perl in general, it's also sometimes helpful to point out that delivery times may be reduced using Perl, as compared to other languages.

If you have a project which has a bottleneck, especially in terms of translation or testing, Perl almost certainly will provide a viable, and quick solution. In conjunction with any persuasion effort, you should not fail to point out that Perl is used, quite extensively, and with extremely reliable and valuable results, at many large computer software and/or hardware companies throughout the world. In fact, many Unix vendors now ship Perl by default, and support is usually just a news-posting away, if you can't find the answer in the *comprehensive* documentation, including this FAQ.

If you face reluctance to upgrading from an older version of perl, then point out that version 4 is utterly unmaintained and unsupported by the Perl Development Team. Another big sell for Perl5 is the large number of modules and extensions which greatly reduce development time for any given task. Also mention that the difference between version 4 and version 5 of Perl is like the difference between awk and C++. (Well, ok, maybe not quite that distinct, but you get the idea.) If you want support and a reasonable guarantee that what you're developing will continue to work in the future, then you have to run the supported version. That probably means running the 5.004 release, although 5.003 isn't that bad (it's just one year and one release behind). Several important bugs were fixed from the 5.000 through 5.002 versions, though, so try upgrading past them if possible.

**AUTHOR AND COPYRIGHT**

**NAME**

perlfaq2 − Obtaining and Learning about Perl ($Revision: 1.24 $, `$Date:` 1998/07/20 23:40:28 $)

**DESCRIPTION**

This section of the FAQ answers questions about where to find source and documentation for Perl, support, and related matters.

### What machines support Perl? Where do I get it?

The standard release of Perl (the one maintained by the perl development team) is distributed only in source code form. You can find this at http://www.perl.com/CPAN/src/latest.tar.gz, which is a gzipped archive in POSIX tar format. This source builds with no porting whatsoever on most Unix systems (Perl's native environment), as well as Plan 9, VMS, QNX, OS/2, and the Amiga.

Although it's rumored that the (imminent) 5.004 release may build on Windows NT, this is yet to be proven. Binary distributions for 32-bit Microsoft systems and for Apple systems can be found http://www.perl.com/CPAN/ports/ directory. Because these are not part of the standard distribution, they may and in fact do differ from the base Perl port in a variety of ways. You'll have to check their respective release notes to see just what the differences are. These differences can be either positive (e.g. extensions for the features of the particular platform that are not supported in the source release of perl) or negative (e.g. might be based upon a less current source release of perl).

A useful FAQ for Win32 Perl users is http://www.endcontsw.com/people/evangelo/Perl_for_Win32_FAQ.html

### How can I get a binary version of Perl?

If you don't have a C compiler because for whatever reasons your vendor did not include one with your system, the best thing to do is grab a binary version of gcc from the net and use that to compile perl with. CPAN only has binaries for systems that are terribly hard to get free compilers for, not for Unix systems.

Your first stop should be http://www.perl.com/CPAN/ports to see what information is already available. A simple installation guide for MS−DOS is available at http://www.cs.ruu.nl/~piet/perl5dos.html , and similarly for Windows 3.1 at http://www.cs.ruu.nl/~piet/perlwin3.html .

### I don't have a C compiler on my system. How can I compile perl?

Since you don't have a C compiler, you're doomed and your vendor should be sacrificed to the Sun gods. But that doesn't help you.

What you need to do is get a binary version of gcc for your system first. Consult the Usenet FAQs for your operating system for information on where to get such a binary version.

### I copied the Perl binary from one machine to another, but scripts don't work.

That's probably because you forgot libraries, or library paths differ. You really should build the whole distribution on the machine it will eventually live on, and then type `make install`. Most other approaches are doomed to failure.

One simple way to check that things are in the right place is to print out the hard-coded `@INC` which perl is looking for.

```
perl -e 'print join("\n",@INC)'
```

If this command lists any paths which don't exist on your system, then you may need to move the appropriate libraries to these locations, or create symlinks, aliases, or shortcuts appropriately.

You might also want to check out the section on *How do I keep my own module/library directory?* in the *perlfaq8* manpage.

### I grabbed the sources and tried to compile but gdbm/dynamic loading/malloc/linking/... failed. How do I make it work?

Read the *INSTALL* file, which is part of the source distribution. It describes in detail how to cope with most idiosyncracies that the Configure script can't work around for any given system or architecture.

### What modules and extensions are available for Perl? What is CPAN? What does CPAN/src/... mean?

CPAN stands for Comprehensive Perl Archive Network, a huge archive replicated on dozens of machines all over the world. CPAN contains source code, non-native ports, documentation, scripts, and many third-party modules and extensions, designed for everything from commercial database interfaces to keyboard/screen control to web walking and CGI scripts. The master machine for CPAN is ftp://ftp.funet.fi/pub/languages/perl/CPAN/, but you can use the address http://www.perl.com/CPAN/CPAN.html to fetch a copy from a "site near you". See http://www.perl.com/CPAN (without a slash at the end) for how this process works.

CPAN/path/... is a naming convention for files available on CPAN sites. CPAN indicates the base directory of a CPAN mirror, and the rest of the path is the path from that directory to the file. For instance, if you're using ftp://ftp.funet.fi/pub/languages/perl/CPAN as your CPAN site, the file CPAN/misc/japh file is downloadable as ftp://ftp.funet.fi/pub/languages/perl/CPAN/misc/japh .

Considering that there are hundreds of existing modules in the archive, one probably exists to do nearly anything you can think of. Current categories under CPAN/modules/by-category/ include perl core modules; development support; operating system interfaces; networking, devices, and interprocess communication; data type utilities; database interfaces; user interfaces; interfaces to other languages; filenames, file systems, and file locking; internationalization and locale; world wide web support; server and daemon utilities; archiving and compression; image manipulation; mail and news; control flow utilities; filehandle and I/O; Microsoft Windows modules; and miscellaneous modules.

### Is there an ISO or ANSI certified version of Perl?

Certainly not. Larry expects that he'll be certified before Perl is.

### Where can I get information on Perl?

The complete Perl documentation is available with the perl distribution. If you have perl installed locally, you probably have the documentation installed as well: type `man perl` if you're on a system resembling Unix. This will lead you to other important man pages, including how to set your `$MANPATH`. If you're not on a Unix system, access to the documentation will be different; for example, it might be only in HTML format. But all proper perl installations have fully-accessible documentation.

You might also try `perldoc perl` in case your system doesn't have a proper man command, or it's been misinstalled. If that doesn't work, try looking in /usr/local/lib/perl5/pod for documentation.

If all else fails, consult the CPAN/doc directory, which contains the complete documentation in various formats, including native pod, troff, html, and plain text. There's also a web page at http://www.perl.com/perl/info/documentation.html that might help.

Many good books have been written about Perl -- see the section below for more details.

### What are the Perl newsgroups on USENET? Where do I post questions?

The now defunct comp.lang.perl newsgroup has been superseded by the following groups:

```
comp.lang.perl.announce           Moderated announcement group
comp.lang.perl.misc               Very busy group about Perl in general
comp.lang.perl.moderated          Moderated discussion group
comp.lang.perl.modules            Use and development of Perl modules
comp.lang.perl.tk                 Using Tk (and X) from Perl

comp.infosystems.www.authoring.cgi  Writing CGI scripts for the Web.
```

Actually, the moderated group hasn't passed yet, but we're keeping our fingers crossed.

There is also USENET gateway to the mailing list used by the crack Perl development team (perl5-porters) at news://news.perl.com/perl.porters-gw/ .

### Where should I post source code?

You should post source code to whichever group is most appropriate, but feel free to cross-post to comp.lang.perl.misc. If you want to cross-post to alt.sources, please make sure it follows their posting standards, including setting the Followup-To header line to NOT include alt.sources; see their FAQ for details.

If you're just looking for software, first use Alta Vista, Deja News, and search CPAN. This is faster and more productive than just posting a request.

### Perl Books

A number of books on Perl and/or CGI programming are available. A few of these are good, some are ok, but many aren't worth your money. Tom Christiansen maintains a list of these books, some with extensive reviews, at http://www.perl.com/perl/critiques/index.html.

The incontestably definitive reference book on Perl, written by the creator of Perl and his apostles, is now in its second edition and fourth printing.

```
Programming Perl (the "Camel Book"):
    Authors: Larry Wall, Tom Christiansen, and Randal Schwartz
    ISBN 1-56592-149-6      (English)
    ISBN 4-89052-384-7      (Japanese)
    (French, German, and Italian translations also available)
```

Note that O'Reilly books are color-coded: turquoise (some would call it teal) covers indicate perl5 coverage, while magenta (some would call it pink) covers indicate perl4 only. Check the cover color before you buy!

If you're already a hard-core systems programmer, then the Camel Book might suffice for you to learn Perl from. But if you're not, check out *Learning Perl* by Randal and Tom. The second edition of "Llama Book" has a blue cover, and is updated for the 5.004 release of Perl.

If you're not an accidental programmer, but a more serious and possibly even degreed computer scientist who doesn't need as much hand-holding as we try to provide in the Llama or its defurred cousin the Gecko, please check out the delightful book, *Perl: The Programmer's Companion*, written by Nigel Chapman.

You can order O'Reilly books directly from O'Reilly & Associates, 1-800-998-9938. Local/overseas is 1-707-829-0515. If you can locate an O'Reilly order form, you can also fax to 1-707-829-0104. See http://www.ora.com/ on the Web.

What follows is a list of the books that the FAQ authors found personally useful. Your mileage may (but, we hope, probably won't) vary.

Recommended books on (or muchly on) Perl are the following. Those marked with a star may be ordered from O'Reilly.

References

```
*Programming Perl
    by Larry Wall, Tom Christiansen, and Randal L. Schwartz

*Perl 5 Desktop Reference
    By Johan Vromans
```

Tutorials *Learning Perl [2nd edition] by Randal L. Schwartz and Tom Christiansen

```
*Learning Perl on Win32 Systems
    by Randal L. Schwartz, Erik Olson, and Tom Christiansen,
        with foreword by Larry Wall

Perl: The Programmer's Companion
    by Nigel Chapman

Cross-Platform Perl
    by Eric F. Johnson

MacPerl: Power and Ease
    by Vicki Brown and Chris Nandor, foreword by Matthias Neeracher
```

Task-Oriented

```
*The Perl Cookbook
    by Tom Christiansen and Nathan Torkington
        with foreword by Larry Wall

Perl5 Interactive Course [2nd edition]
    by Jon Orwant

*Advanced Perl Programming
    by Sriram Srinivasan

Effective Perl Programming
    by Joseph Hall
```

Special Topics

```
*Mastering Regular Expressions
    by Jeffrey Friedl

How to Set up and Maintain a World Wide Web Site [2nd edition]
    by Lincoln Stein
```

**Perl in Magazines**

The first and only periodical devoted to All Things Perl, *The Perl Journal* contains tutorials, demonstrations, case studies, announcements, contests, and much more. TPJ has columns on web development, databases, Win32 Perl, graphical programming, regular expressions, and networking, and sponsors the Obfuscated Perl Contest. It is published quarterly by Jon Orwant. See http://www.tpj.com/ or send mail to subscriptions@tpj.com.

Beyond this, magazines that frequently carry high-quality articles on Perl are *Web Techniques* (see http://www.webtechniques.com/), *Performance Computing* (http://www.performance-computing.com/), and Usenix's newsletter/magazine to its members, *login:*, at http://www.usenix.org/. Randal's Web Technique's columns are available on the web at http://www.stonehenge.com/merlyn/WebTechniques/.

### Perl on the Net: FTP and WWW Access

To get the best (and possibly cheapest) performance, pick a site from the list below and use it to grab the complete list of mirror sites. From there you can find the quickest site for you. Remember, the following list is *not* the complete list of CPAN mirrors.

```
http://www.perl.com/CPAN      (redirects to another mirror)
http://www.perl.org/CPAN
ftp://ftp.funet.fi/pub/languages/perl/CPAN/
http://www.cs.ruu.nl/pub/PERL/CPAN/
ftp://ftp.cs.colorado.edu/pub/perl/CPAN/
```

http:/www.oasis.leo.org/perl/ has, amongst other things, source to versions 1 through 5 of Perl.

### What mailing lists are there for perl?

Most of the major modules (tk, CGI, libwww-perl) have their own mailing lists. Consult the documentation that came with the module for subscription information. The following are a list of mailing lists related to perl itself.

If you subscribe to a mailing list, it behooves you to know how to unsubscribe from it. Strident pleas to the list itself to get you off will not be favorably received.

MacPerl

There is a mailing list for discussing Macintosh Perl. Contact ''mac-perl-request@iis.ee.ethz.ch''.

Also see Matthias Neeracher's (the creator and maintainer of MacPerl) webpage at http://www.iis.ee.ethz.ch/~neeri/macintosh/perl.html for many links to interesting MacPerl sites, and the applications/MPW tools, precompiled.

Perl5-Porters

The core development team have a mailing list for discussing fixes and changes to the language. Send mail to ''perl5-porters-request@perl.org'' with help in the body of the message for information on subscribing.

NTPerl

This list is used to discuss issues involving Win32 Perl 5 (Windows NT and Win95). Subscribe by mailing ListManager@ActiveWare.com with the message body:

```
subscribe Perl-Win32-Users
```

The list software, also written in perl, will automatically determine your address, and subscribe you automatically. To unsubscribe, mail the following in the message body to the same address like so:

```
unsubscribe Perl-Win32-Users
```

You can also check http://www.activeware.com/ and select ''Mailing Lists'' to join or leave this list.

Perl-Packrats

Discussion related to archiving of perl materials, particularly the Comprehensive Perl Archive Network (CPAN). Subscribe by emailing majordomo@cis.ufl.edu:

```
subscribe perl-packrats
```

The list software, also written in perl, will automatically determine your address, and subscribe you automatically. To unsubscribe, simple prepend the same command with an ''un'', and mail to the same address like so:

```
unsubscribe perl-packrats
```

**Archives of comp.lang.perl.misc**

Have you tried Deja News or Alta Vista?

ftp.cis.ufl.edu:/pub/perl/comp.lang.perl.*/monthly has an almost complete collection dating back to 12/89 (missing 08/91 through 12/93). They are kept as one large file for each month.

You'll probably want more a sophisticated query and retrieval mechanism than a file listing, preferably one that allows you to retrieve articles using a fast-access indices, keyed on at least author, date, subject, thread (as in "trn") and probably keywords. The best solution the FAQ authors know of is the MH pick command, but it is very slow to select on 18000 articles.

If you have, or know where can be found, the missing sections, please let perlfaq-suggestions@perl.com know.

**Where can I buy a commercial version of Perl?**

In a sense, Perl already *is* commercial software: It has a licence that you can grab and carefully read to your manager. It is distributed in releases and comes in well-defined packages. There is a very large user community and an extensive literature. The comp.lang.perl.* newsgroups and several of the mailing lists provide free answers to your questions in near real-time. Perl has traditionally been supported by Larry, dozens of software designers and developers, and thousands of programmers, all working for free to create a useful thing to make life better for everyone.

However, these answers may not suffice for managers who require a purchase order from a company whom they can sue should anything go wrong. Or maybe they need very serious hand-holding and contractual obligations. Shrink-wrapped CDs with perl on them are available from several sources if that will help.

Or you can purchase a real support contract. Although Cygnus historically provided this service, they no longer sell support contracts for Perl. Instead, the Paul Ingram Group will be taking up the slack through The Perl Clinic. The following is a commercial from them:

"Do you need professional support for Perl and/or Oraperl? Do you need a support contract with defined levels of service? Do you want to pay only for what you need?

"The Paul Ingram Group has provided quality software development and support services to some of the world's largest corporations for ten years. We are now offering the same quality support services for Perl at The Perl Clinic. This service is led by Tim Bunce, an active perl porter since 1994 and well known as the author and maintainer of the DBI, DBD::Oracle, and Oraperl modules and author/co-maintainer of The Perl 5 Module List. We also offer Oracle users support for Perl5 Oraperl and related modules (which Oracle is planning to ship as part of Oracle Web Server 3). 20% of the profit from our Perl support work will be donated to The Perl Institute."

For more information, contact the The Perl Clinic:

```
Tel:    +44 1483 424424
Fax:    +44 1483 419419
Web:    http://www.perl.co.uk/
Email:  perl-support-info@perl.co.uk or Tim.Bunce@ig.co.uk
```

**Where do I send bug reports?**

If you are reporting a bug in the perl interpreter or the modules shipped with perl, use the *perlbug* program in the perl distribution or mail your report to perlbug@perl.com.

If you are posting a bug with a non-standard port (see the answer to "What platforms is Perl available for?"), a binary distribution, or a non-standard module (such as Tk, CGI, etc), then please see the documentation that came with it to determine the correct place to post bugs.

Read the *perlbug* (1) man page (perl5.004 or later) for more information.

### What is perl.com? perl.org? The Perl Institute?

The perl.com domain is Tom Christiansen's domain. He created it as a public service long before perl.org came about. Despite the name, it's a pretty non-commercial site meant to be a clearinghouse for information about all things Perlian, accepting no paid advertisements, bouncy happy gifs, or silly java applets on its pages. The Perl Home Page at http://www.perl.com/ is currently hosted on a T3 line courtesy of Songline Systems, a software-oriented subsidiary of O'Reilly and Associates.

perl.org is the official vehicle for The Perl Institute. The motto of TPI is ''helping people help Perl help people'' (or something like that). It's a non-profit organization supporting development, documentation, and dissemination of perl. Current directors of TPI include Larry Wall, Tom Christiansen, and Randal Schwartz, whom you may have heard of somewhere else around here.

### How do I learn about object-oriented Perl programming?

the *perltoot* manpage (distributed with 5.004 or later) is a good place to start. Also, the *perlobj* manpage, the *perlref* manpage, and the *perlmod* manpage are useful references, while the *perlbot* manpage has some excellent tips and tricks.

## AUTHOR AND COPYRIGHT



When included as an integrated part of the Standard Distribution of Perl or of its documentation (printed or otherwise), this works is covered under Perl's Artistic Licence. For separate distributions of all or part of this FAQ outside of that, see the *perlfaq* manpage.

Irrespective of its distribution, all code examples here are public domain. You are permitted and encouraged to use this code and any derivatives thereof in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit to the FAQ would be courteous but is not required.

**NAME**

perlfaq3 − Programming Tools ($Revision: 1.28 $, `$Date:` 1998/07/16 22:08:49 $)

**DESCRIPTION**

This section of the FAQ answers questions related to programmer tools and programming support.

### How do I do (anything)?

Have you looked at CPAN (see the *perlfaq2* manpage)? The chances are that someone has already written a module that can solve your problem. Have you read the appropriate man pages? Here's a brief index:

```
Basics          perldata, perlvar, perlsyn, perlop, perlsub
Execution       perlrun, perldebug
Functions       perlfunc
Objects         perlref, perlmod, perlobj, perltie
Data Structures perlref, perllol, perldsc
Modules         perlmod, perlmodlib, perlsub
Regexps         perlre, perlfunc, perlop, perllocale
Moving to perl5 perltrap, perl
Linking w/C     perlxstut, perlxs, perlcall, perlguts, perlembed
Various         http://www.perl.com/CPAN/doc/FMTEYEWTK/index.html
                (not a man-page but still useful)
```

the *perltoc* manpage provides a crude table of contents for the perl man page set.

### How can I use Perl interactively?

The typical approach uses the Perl debugger, described in the *perldebug* (1) man page, on an "empty" program, like this:

```
perl -de 42
```

Now just type in any legal Perl code, and it will be immediately evaluated. You can also examine the symbol table, get stack backtraces, check variable values, set breakpoints, and other operations typically found in symbolic debuggers.

### Is there a Perl shell?

In general, no. The Shell.pm module (distributed with perl) makes perl try commands which aren't part of the Perl language as shell commands. perlsh from the source distribution is simplistic and uninteresting, but may still be what you want.

### How do I debug my Perl programs?

Have you used `-w`? It enables warnings for dubious practices.

Have you tried `use strict`? It prevents you from using symbolic references, makes you predeclare any subroutines that you call as bare words, and (probably most importantly) forces you to predeclare your variables with `my` or `use vars`.

Did you check the returns of each and every system call? The operating system (and thus Perl) tells you whether they worked or not, and if not why.

```
open(FH, "> /etc/cantwrite")
  or die "Couldn't write to /etc/cantwrite: $!\n";
```

Did you read the *perltrap* manpage? It's full of gotchas for old and new Perl programmers, and even has sections for those of you who are upgrading from languages like *awk* and *C*.

Have you tried the Perl debugger, described in the *perldebug* manpage? You can step through your program and see what it's doing and thus work out why what it's doing isn't what it should be doing.

### How do I profile my Perl programs?

You should get the Devel::DProf module from CPAN, and also use Benchmark.pm from the standard distribution. Benchmark lets you time specific portions of your code, while Devel::DProf gives detailed breakdowns of where your code spends its time.

Here's a sample use of Benchmark:

```
use Benchmark;

@junk = `cat /etc/motd`;
$count = 10_000;

timethese($count, {
          'map' => sub { my @a = @junk;
                         map { s/a/b/ } @a;
                         return @a
                       },
          'for' => sub { my @a = @junk;
                         local $_;
                         for (@a) { s/a/b/ };
                         return @a },
         });
```

This is what it prints (on one machine—your results will be dependent on your hardware, operating system, and the load on your machine):

```
Benchmark: timing 10000 iterations of for, map...
       for:  4 secs ( 3.97 usr  0.01 sys =  3.98 cpu)
       map:  6 secs ( 4.97 usr  0.00 sys =  4.97 cpu)
```

### How do I cross-reference my Perl programs?

The B::Xref module, shipped with the new, alpha-release Perl compiler (not the general distribution prior to the 5.005 release), can be used to generate cross-reference reports for Perl programs.

```
perl -MO=Xref[,OPTIONS] scriptname.plx
```

### Is there a pretty-printer (formatter) for Perl?

There is no program that will reformat Perl as much as *indent*(1) does for C. The complex feedback between the scanner and the parser (this feedback is what confuses the vgrind and emacs programs) makes it challenging at best to write a stand-alone Perl parser.

Of course, if you simply follow the guidelines in the *perlstyle* manpage, you shouldn't need to reformat. The habit of formatting your code as you write it will help prevent bugs. Your editor can and should help you with this. The perl-mode for emacs can provide a remarkable amount of help with most (but not all) code, and even less programmable editors can provide significant assistance.

If you are used to using *vgrind* program for printing out nice code to a laser printer, you can take a stab at this using http://www.perl.com/CPAN/doc/misc/tips/working.vgrind.entry, but the results are not particularly satisfying for sophisticated code.

### Is there a ctags for Perl?

There's a simple one at http://www.perl.com/CPAN/authors/id/TOMC/scripts/ptags.gz which may do the trick.

### Where can I get Perl macros for vi?

For a complete version of Tom Christiansen's vi configuration file, see http://www.perl.com/CPAN/authors/Tom_Christiansen/scripts/toms.exrc, the standard benchmark file for vi emulators. This runs best with nvi, the current version of vi out of Berkeley, which incidentally can be built with an embedded Perl interpreter -- see http://www.perl.com/CPAN/src/misc.

### Where can I get perl-mode for emacs?

Since Emacs version 19 patchlevel 22 or so, there have been both a perl-mode.el and support for the perl debugger built in. These should come with the standard Emacs 19 distribution.

In the perl source directory, you'll find a directory called ''emacs'', which contains a cperl-mode that color-codes keywords, provides context-sensitive help, and other nifty things.

Note that the perl-mode of emacs will have fits with `"main'foo"` (single quote), and mess up the indentation and hilighting. You should be using `"main::foo"` in new Perl code anyway, so this shouldn't be an issue.

### How can I use curses with Perl?

The Curses module from CPAN provides a dynamically loadable object module interface to a curses library. A small demo can be found at the directory http://www.perl.com/CPAN/authors/Tom_Christiansen/scripts/rep; this program repeats a command and updates the screen as needed, rendering **rep ps axu** similar to **top**.

### How can I use X or Tk with Perl?

Tk is a completely Perl-based, object-oriented interface to the Tk toolkit that doesn't force you to use Tcl just to get at Tk. Sx is an interface to the Athena Widget set. Both are available from CPAN. See the directory http://www.perl.com/CPAN/modules/by-category/08_User_Interfaces/

Invaluable for Perl/Tk programming are: the Perl/Tk FAQ at http://w4.lns.cornell.edu/~pvhp/ptk/ptkTOC.html , the Perl/Tk Reference Guide available at http://www.perl.com/CPAN−local/authors/Stephen_O_Lidie/ , and the online manpages at http://www-users.cs.umn.edu/~amundson/perl/perltk/toc.html .

### How can I generate simple menus without using CGI or Tk?

The http://www.perl.com/CPAN/authors/id/SKUNZ/perlmenu.v4.0.tar.gz module, which is curses-based, can help with this.

### What is undump?

See the next questions.

### How can I make my Perl program run faster?

The best way to do this is to come up with a better algorithm. This can often make a dramatic difference. Chapter 8 in the Camel has some efficiency tips in it you might want to look at. Jon Bentley's book ''Programming Pearls'' (that's not a misspelling!) has some good tips on optimization, too. Advice on benchmarking boils down to: benchmark and profile to make sure you're optimizing the right part, look for better algorithms instead of microtuning your code, and when all else fails consider just buying faster hardware.

A different approach is to autoload seldom-used Perl code. See the AutoSplit and AutoLoader modules in the standard distribution for that. Or you could locate the bottleneck and think about writing just that part in C, the way we used to take bottlenecks in C code and write them in assembler. Similar to rewriting in C is the use of modules that have critical sections written in C (for instance, the PDL module from CPAN).

In some cases, it may be worth it to use the backend compiler to produce byte code (saving compilation time) or compile into C, which will certainly save compilation time and sometimes a small amount (but not much) execution time. See the question about compiling your Perl programs for more on the compiler— the wins aren't as obvious as you'd hope.

If you're currently linking your perl executable to a shared *libc.so*, you can often gain a 10-25% performance benefit by rebuilding it to link with a static libc.a instead. This will make a bigger perl executable, but your Perl programs (and programmers) may thank you for it. See the *INSTALL* file in the source distribution for more information.

Unsubstantiated reports allege that Perl interpreters that use sfio outperform those that don't (for IO intensive applications). To try this, see the *INSTALL* file in the source distribution, especially the "Selecting File IO mechanisms" section.

The undump program was an old attempt to speed up your Perl program by storing the already-compiled form to disk. This is no longer a viable option, as it only worked on a few architectures, and wasn't a good solution anyway.

### How can I make my Perl program take less memory?

When it comes to time-space tradeoffs, Perl nearly always prefers to throw memory at a problem. Scalars in Perl use more memory than strings in C, arrays take more that, and hashes use even more. While there's still a lot to be done, recent releases have been addressing these issues. For example, as of 5.004, duplicate hash keys are shared amongst all hashes using them, so require no reallocation.

In some cases, using *substr()* or *vec()* to simulate arrays can be highly beneficial. For example, an array of a thousand booleans will take at least 20,000 bytes of space, but it can be turned into one 125-byte bit vector for a considerable memory savings. The standard Tie::SubstrHash module can also help for certain types of data structure. If you're working with specialist data structures (matrices, for instance) modules that implement these in C may use less memory than equivalent Perl modules.

Another thing to try is learning whether your Perl was compiled with the system malloc or with Perl's builtin malloc. Whichever one it is, try using the other one and see whether this makes a difference. Information about malloc is in the *INSTALL* file in the source distribution. You can find out whether you are using perl's malloc by typing `perl -V:usemymalloc`.

### Is it unsafe to return a pointer to local data?

No, Perl's garbage collection system takes care of this.

```
sub makeone {
    my @a = ( 1 .. 10 );
    return \@a;
}

for $i ( 1 .. 10 ) {
    push @many, makeone();
}

print $many[4][5], "\n";

print "@many\n";
```

### How can I free an array or hash so my program shrinks?

You can't. On most operating systems, memory allocated to a program can never be returned to the system. That's why long-running programs sometimes re-exec themselves. Some operating systems (notably, FreeBSD) allegedly reclaim large chunks of memory that is no longer used, but it doesn't appear to happen with Perl (yet). The Mac appears to be the only platform that will reliably (albeit, slowly) return memory to the OS.

However, judicious use of *my()* on your variables will help make sure that they go out of scope so that Perl can free up their storage for use in other parts of your program. A global variable, of course, never goes out of scope, so you can't get its space automatically reclaimed, although *undef()*ing and/or *delete()*ing it will achieve the same effect. In general, memory allocation and de-allocation isn't something you can or should be worrying about much in Perl, but even this capability (preallocation of data types) is in the works.

### How can I make my CGI script more efficient?

Beyond the normal measures described to make general Perl programs faster or smaller, a CGI program has additional issues. It may be run several times per second. Given that each time it runs it will need to be re-compiled and will often allocate a megabyte or more of system memory, this can be a killer. Compiling into C **isn't going to help you** because the process start-up overhead is where the bottleneck is.

There are two popular ways to avoid this overhead. One solution involves running the Apache HTTP server (available from http://www.apache.org/) with either of the mod_perl or mod_fastcgi plugin modules.

With mod_perl and the Apache::Registry module (distributed with mod_perl), httpd will run with an embedded Perl interpreter which pre-compiles your script and then executes it within the same address space without forking. The Apache extension also gives Perl access to the internal server API, so modules written in Perl can do just about anything a module written in C can. For more on mod_perl, see http://perl.apache.org/

With the FCGI module (from CPAN), a Perl executable compiled with sfio (see the *INSTALL* file in the distribution) and the mod_fastcgi module (available from http://www.fastcgi.com/) each of your perl scripts becomes a permanent CGI daemon process.

Both of these solutions can have far-reaching effects on your system and on the way you write your CGI scripts, so investigate them with care.

See http://www.perl.com/CPAN/modules/by-category/15_World_Wide_Web_HTML_HTTP_CGI/ .

A non-free, commerical product, ''The Velocity Engine for Perl'', (http://www.binevolve.com/ or http://www.binevolve.com/bine/vep) might also be worth looking at. It will allow you to increase the performance of your perl scripts, upto 25 times faster than normal CGI perl by running in persistent perl mode, or 4 to 5 times faster without any modification to your existing CGI scripts. Fully functional evaluation copies are available from the web site.

### How can I hide the source for my Perl program?

Delete it. :-) Seriously, there are a number of (mostly unsatisfactory) solutions with varying levels of ''security''.

First of all, however, you *can't* take away read permission, because the source code has to be readable in order to be compiled and interpreted. (That doesn't mean that a CGI script's source is readable by people on the web, though, only by people with access to the filesystem) So you have to leave the permissions at the socially friendly 0755 level.

Some people regard this as a security problem. If your program does insecure things, and relies on people not knowing how to exploit those insecurities, it is not secure. It is often possible for someone to determine the insecure things and exploit them without viewing the source. Security through obscurity, the name for hiding your bugs instead of fixing them, is little security indeed.

You can try using encryption via source filters (Filter::* from CPAN), but crackers might be able to decrypt it. You can try using the byte code compiler and interpreter described below, but crackers might be able to de-compile it. You can try using the native-code compiler described below, but crackers might be able to disassemble it. These pose varying degrees of difficulty to people wanting to get at your code, but none can definitively conceal it (this is true of every language, not just Perl).

If you're concerned about people profiting from your code, then the bottom line is that nothing but a restrictive licence will give you legal security. License your software and pepper it with threatening statements like ''This is unpublished proprietary software of XYZ Corp. Your access to it does not give you permission to use it blah blah blah.'' We are not lawyers, of course, so you should see a lawyer if you want to be sure your licence's wording will stand up in court.

**How can I compile my Perl program into byte code or C?**

Malcolm Beattie has written a multifunction backend compiler, available from CPAN, that can do both these things. It is as of Jul-1998 in late alpha release, which means it's fun to play with if you're a programmer but not really for people looking for turn-key solutions.

Merely compiling into C does not in and of itself guarantee that your code will run very much faster. That's because except for lucky cases where a lot of native type inferencing is possible, the normal Perl run time system is still present and so your program will take just as long to run and be just as big. Most programs save little more than compilation time, leaving execution no more than 10-30% faster. A few rare programs actually benefit significantly (like several times faster), but this takes some tweaking of your code.

The 5.005 release of Perl itself, whose main goal is merging the various non-Unix ports back into the one Perl source, will also have preliminary (strictly beta) support for Malcolm's compiler and his light-weight processes (sometimes called "threads").

You'll probably be astonished to learn that the current version of the compiler generates a compiled form of your script whose executable is just as big as the original perl executable, and then some. That's because as currently written, all programs are prepared for a full *eval()* statement. You can tremendously reduce this cost by building a shared *libperl.so* library and linking against that. See the *INSTALL* podfile in the perl source distribution for details. If you link your main perl binary with this, it will make it miniscule. For example, on one author's system, */usr/bin/perl* is only 11k in size!

In general, the compiler will do nothing to make a Perl program smaller, faster, more portable, or more secure. In fact, it will usually hurt all of those. The executable will be bigger, your VM system may take longer to load the whole thing, the binary is fragile and hard to fix, and compilation never stopped software piracy in the form of crackers, viruses, or bootleggers. The real advantage of the compiler is merely packaging, and once you see the size of what it makes (well, unless you use a shared *libperl.so*), you'll probably want a complete Perl install anywayt.

**How can I get** `#!perl` to work on [MS−DOS,NT,...]?

For OS/2 just use

```
extproc perl −S −your_switches
```

as the first line in `*.cmd` file (`−S` due to a bug in cmd.exe's 'extproc' handling). For DOS one should first invent a corresponding batch file, and codify it in `ALTERNATIVE_SHEBANG` (see the *INSTALL* file in the source distribution for more information).

The Win95/NT installation, when using the ActiveState port of Perl, will modify the Registry to associate the `.pl` extension with the perl interpreter. If you install another port (Gurusaramy Sarathy's is the recommended Win95/NT port), or (eventually) build your own Win95/NT Perl using WinGCC, then you'll have to modify the Registry yourself.

Macintosh perl scripts will have the the appropriate Creator and Type, so that double-clicking them will invoke the perl application.

*IMPORTANT!*: Whatever you do, PLEASE don't get frustrated, and just throw the perl interpreter into your cgi-bin directory, in order to get your scripts working for a web server. This is an EXTREMELY big security risk. Take the time to figure out how to do it correctly.

**Can I write useful perl programs on the command line?**

Yes. Read the *perlrun* manpage for more information. Some examples follow. (These assume standard Unix shell quoting rules.)

```
# sum first and last fields
perl −lane 'print $F[0] + $F[−1]' *
```

```
# identify text files
perl -le 'for(@ARGV) {print if -f && -T _}' *

# remove (most) comments from C program
perl -0777 -pe 's{/\*.*?\*/}{}gs' foo.c

# make file a month younger than today, defeating reaper daemons
perl -e '$X=24*60*60; utime(time(),time() + 30 * $X,@ARGV)' *

# find first unused uid
perl -le '$i++ while getpwuid($i); print $i'

# display reasonable manpath
echo $PATH | perl -nl -072 -e '
    s![^/+]*$!man!&&-d&&!$s{$_}++&&push@m,$_;END{print"@m"}'
```

Ok, the last one was actually an obfuscated perl entry. :–)

**Why don't perl one-liners work on my DOS/Mac/VMS system?**

The problem is usually that the command interpreters on those systems have rather different ideas about quoting than the Unix shells under which the one-liners were created. On some systems, you may have to change single-quotes to double ones, which you must *NOT* do on Unix or Plan9 systems. You might also have to change a single % to a %%.

For example:

```
# Unix
perl -e 'print "Hello world\n"'

# DOS, etc.
perl -e "print \"Hello world\n\""

# Mac
print "Hello world\n"
 (then Run "Myscript" or Shift-Command-R)

# VMS
perl -e "print ""Hello world\n"""
```

The problem is that none of this is reliable: it depends on the command interpreter. Under Unix, the first two often work. Under DOS, it's entirely possible neither works. If 4DOS was the command shell, you'd probably have better luck like this:

```
perl -e "print <Ctrl-x>"Hello world\n<Ctrl-x>""
```

Under the Mac, it depends which environment you are using. The MacPerl shell, or MPW, is much like Unix shells in its support for several quoting variants, except that it makes free use of the Mac's non-ASCII characters as control characters.

There is no general solution to all of this. It is a mess, pure and simple. Sucks to be away from Unix, huh? :–)

[Some of this answer was contributed by Kenneth Albanowski.]

### Where can I learn about CGI or Web programming in Perl?

For modules, get the CGI or LWP modules from CPAN. For textbooks, see the two especially dedicated to web stuff in the question on books. For problems and questions related to the web, like "Why do I get 500 Errors" or "Why doesn't it run from the browser right when it runs fine on the command line", see these sources:

```
WWW Security FAQ
    http://www.w3.org/Security/Faq/

Web FAQ
    http://www.boutell.com/faq/

CGI FAQ
    http://www.webthing.com/page.cgi/cgifaq

HTTP Spec
    http://www.w3.org/pub/WWW/Protocols/HTTP/

HTML Spec
    http://www.w3.org/TR/REC-html40/
    http://www.w3.org/pub/WWW/MarkUp/

CGI Spec
    http://www.w3.org/CGI/

CGI Security FAQ
    http://www.go2net.com/people/paulp/cgi-security/safe-cgi.txt
```

### Where can I learn about object-oriented Perl programming?

the *perltoot* manpage is a good place to start, and you can use the *perlobj* manpage and the *perlbot* manpage for reference. Perltoot didn't come out until the 5.004 release, but you can get a copy (in pod, html, or postscript) from http://www.perl.com/CPAN/doc/FMTEYEWTK/ .

### Where can I learn about linking C with Perl? [h2xs, xsubpp]

If you want to call C from Perl, start with the *perlxstut* manpage, moving on to the *perlxs* manpage, the *xsubpp* manpage, and the *perlguts* manpage. If you want to call Perl from C, then read the *perlembed* manpage, the *perlcall* manpage, and the *perlguts* manpage. Don't forget that you can learn a lot from looking at how the authors of existing extension modules wrote their code and solved their problems.

### I've read perlembed, perlguts, etc., but I can't embed perl in my C program, what am I doing wrong?

Download the ExtUtils::Embed kit from CPAN and run 'make test'. If the tests pass, read the pods again and again and again. If they fail, see the *perlbug* manpage and send a bugreport with the output of `make test TEST_VERBOSE=1` along with `perl -V`.

### When I tried to run my script, I got this message. What does it mean?

the *perldiag* manpage has a complete list of perl's error messages and warnings, with explanatory text. You can also use the splain program (distributed with perl) to explain the error messages:

```
perl program 2>diag.out
splain [-v] [-p] diag.out
```

or change your program to explain the messages for you:

```
use diagnostics;
```

or

```
use diagnostics -verbose;
```

### What's MakeMaker?

This module (part of the standard perl distribution) is designed to write a Makefile for an extension module from a Makefile.PL. For more information, see the *ExtUtils::MakeMaker* manpage.

## AUTHOR AND COPYRIGHT

## NAME

perlfaq4 − Data Manipulation ($Revision: 1.25 $, `$Date:` 1998/07/16 22:49:55 $)

## DESCRIPTION

The section of the FAQ answers question related to the manipulation of data as numbers, dates, strings, arrays, hashes, and miscellaneous data issues.

## Data: Numbers

### Why am I getting long decimals (eg, 19.9499999999999) instead of the numbers I should be getting (eg, 19.95)?

The infinite set that a mathematician thinks of as the real numbers can only be approximate on a computer, since the computer only has a finite number of bits to store an infinite number of, um, numbers.

Internally, your computer represents floating-point numbers in binary. Floating-point numbers read in from a file or appearing as literals in your program are converted from their decimal floating-point representation (eg, 19.95) to the internal binary representation.

However, 19.95 can't be precisely represented as a binary floating-point number, just like 1/3 can't be exactly represented as a decimal floating-point number. The computer's binary representation of 19.95, therefore, isn't exactly 19.95.

When a floating-point number gets printed, the binary floating-point representation is converted back to decimal. These decimal numbers are displayed in either the format you specify with *printf()*, or the current output format for numbers (see the section on *$#* in the *perlvar* manpage if you use print. `$#` has a different default value in Perl5 than it did in Perl4. Changing `$#` yourself is deprecated.

This affects **all** computer languages that represent decimal floating-point numbers in binary, not just Perl. Perl provides arbitrary-precision decimal numbers with the Math::BigFloat module (part of the standard Perl distribution), but mathematical operations are consequently slower.

To get rid of the superfluous digits, just use a format (eg, `printf("%.2f", 19.95)`) to get the required precision. See the section on *Floating-point Arithmetic* in the *perlop* manpage.

### Why isn't my octal data interpreted correctly?

Perl only understands octal and hex numbers as such when they occur as literals in your program. If they are read in from somewhere and assigned, no automatic conversion takes place. You must explicitly use *oct()* or *hex()* if you want the values converted. *oct()* interprets both hex ("0x350") numbers and octal ones ("0350'' or even without the leading "0'', like "377"), while *hex()* only converts hexadecimal ones, with or without a leading "0x'', like "0x255'', "3A'', "ff'', or "deadbeef''.

This problem shows up most often when people try using *chmod()*, *mkdir()*, *umask()*, or *sysopen()*, which all want permissions in octal.

```
chmod(644,  $file); # WRONG -- perl -w catches this
chmod(0644, $file); # right
```

### Does perl have a round function? What about *ceil()* and *floor()*? Trig functions?

Remember that *int()* merely truncates toward 0. For rounding to a certain number of digits, *sprintf()* or *printf()* is usually the easiest route.

```
printf("%.3f", 3.1415926535);       # prints 3.142
```

The POSIX module (part of the standard perl distribution) implements *ceil()*, *floor()*, and a number of other mathematical and trigonometric functions.

```
use POSIX;
$ceil   = ceil(3.5);                        # 4
$floor  = floor(3.5);                       # 3
```

In 5.000 to 5.003 Perls, trigonometry was done in the Math::Complex module. With 5.004, the Math::Trig module (part of the standard perl distribution) implements the trigonometric functions. Internally it uses the Math::Complex module and some functions can break out from the real axis into the complex plane, for example the inverse sine of 2.

Rounding in financial applications can have serious implications, and the rounding method used should be specified precisely. In these cases, it probably pays not to trust whichever system rounding is being used by Perl, but to instead implement the rounding function you need yourself.

### How do I convert bits into ints?

To turn a string of 1s and 0s like `10110110` into a scalar containing its binary value, use the *pack()* function (documented in the section on *pack* in the *perlfunc* manpage):

```
$decimal = pack('B8', '10110110');
```

Here's an example of going the other way:

```
$binary_string = join('', unpack('B*', "\x29"));
```

### How do I multiply matrices?

Use the Math::Matrix or Math::MatrixReal modules (available from CPAN) or the PDL extension (also available from CPAN).

### How do I perform an operation on a series of integers?

To call a function on each element in an array, and collect the results, use:

```
@results = map { my_func($_) } @array;
```

For example:

```
@triple = map { 3 * $_ } @single;
```

To call a function on each element of an array, but ignore the results:

```
foreach $iterator (@array) {
    &my_func($iterator);
}
```

To call a function on each integer in a (small) range, you **can** use:

```
@results = map { &my_func($_) } (5 .. 25);
```

but you should be aware that the `..` operator creates an array of all integers in the range. This can take a lot of memory for large ranges. Instead use:

```
@results = ();
for ($i=5; $i < 500_005; $i++) {
    push(@results, &my_func($i));
}
```

### How can I output Roman numerals?

Get the http://www.perl.com/CPAN/modules/by-module/Roman module.

### Why aren't my random numbers random?

The short explanation is that you're getting pseudorandom numbers, not random ones, because computers are good at being predictable and bad at being random (despite appearances caused by bugs in your programs :–). A longer explanation is available on http://www.perl.com/CPAN/doc/FMTEYEWTK/random, courtesy of Tom Phoenix. John von Neumann said, "Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin."

You should also check out the Math::TrulyRandom module from CPAN. It uses the imperfections in your system's timer to generate random numbers, but this takes quite a while. If you want a better pseudorandom generator than comes with your operating system, look at "Numerical Recipes in C" at http://nr.harvard.edu/nr/bookc.html .

## Data: Dates

### How do I find the week-of-the-year/day-of-the-year?

The day of the year is in the array returned by *localtime()* (see the section on *localtime* in the *perlfunc* manpage):

```
$day_of_year = (localtime(time()))[7];
```

or more legibly (in 5.004 or higher):

```
use Time::localtime;
$day_of_year = localtime(time())->yday;
```

You can find the week of the year by dividing this by 7:

```
$week_of_year = int($day_of_year / 7);
```

Of course, this believes that weeks start at zero. The Date::Calc module from CPAN has a lot of date calculation functions, including day of the year, week of the year, and so on.

### How can I compare two dates and find the difference?

If you're storing your dates as epoch seconds then simply subtract one from the other. If you've got a structured date (distinct year, day, month, hour, minute, seconds values) then use one of the Date::Manip and Date::Calc modules from CPAN.

### How can I take a string and turn it into epoch seconds?

If it's a regular enough string that it always has the same format, you can split it up and pass the parts to `timelocal` in the standard Time::Local module. Otherwise, you should look into the Date::Calc and Date::Manip modules from CPAN.

### How can I find the Julian Day?

Neither Date::Manip nor Date::Calc deal with Julian days. Instead, there is an example of Julian date calculation that should help you in http://www.perl.com/CPAN/authors/David_Muir_Sharnoff/modules/Time/JulianDay.pm.gz .

### Does Perl have a year 2000 problem? Is Perl Y2K compliant?

Short answer: No, Perl does not have a Year 2000 problem. Yes, Perl is Y2K compliant.

Long answer: Perl is just as Y2K compliant as your pencil—no more, and no less. The date and time functions supplied with perl (gmtime and localtime) supply adequate information to determine the year well beyond 2000 (2038 is when trouble strikes for 32-bit machines). The year returned by these functions when used in an array context is the year minus 1900. For years between 1910 and 1999 this *happens* to be a 2-digit decimal number. To avoid the year 2000 problem simply do not treat the year as a 2-digit number. It isn't.

When *gmtime()* and *localtime()* are used in scalar context they return a timestamp string that contains a fully-expanded year. For example, `$timestamp = gmtime(1005613200)` sets `$timestamp` to "Tue Nov 13 01:00:00 2001". There's no year 2000 problem here.

That doesn't mean that Perl can't be used to create non-Y2K compliant programs. It can. But so can your pencil. It's the fault of the user, not the language. At the risk of inflaming the NRA: "Perl doesn't break Y2K, people do." See http://language.perl.com/news/y2k.html for a longer exposition.

## Data: Strings

### How do I validate input?

The answer to this question is usually a regular expression, perhaps with auxiliary logic. See the more specific questions (numbers, mail addresses, etc.) for details.

### How do I unescape a string?

It depends just what you mean by "escape". URL escapes are dealt with in the *perlfaq9* manpage. Shell escapes with the backslash (\) character are removed with:

```
s/\\(.)/$1/g;
```

This won't expand `"\n"` or `"\t"` or any other special escapes.

### How do I remove consecutive pairs of characters?

To turn `"abbcccd"` into `"abccd"`:

```
s/(.)\1/$1/g;
```

### How do I expand function calls in a string?

This is documented in the *perlref* manpage. In general, this is fraught with quoting and readability problems, but it is possible. To interpolate a subroutine call (in list context) into a string:

```
print "My sub returned @{[mysub(1,2,3)]} that time.\n";
```

If you prefer scalar context, similar chicanery is also useful for arbitrary expressions:

```
print "That yields ${\($n + 5)} widgets\n";
```

Version 5.004 of Perl had a bug that gave list context to the expression in `${...}`, but this is fixed in version 5.005.

See also "How can I expand variables in text strings?" in this section of the FAQ.

### How do I find matching/nesting anything?

This isn't something that can be done in one regular expression, no matter how complicated. To find something between two single characters, a pattern like `/x([^x]*)x/` will get the intervening bits in `$1`. For multiple ones, then something more like `/alpha(.*?)omega/` would be needed. But none of these deals with nested patterns, nor can they. For that you'll have to write a parser.

If you are serious about writing a parser, there are a number of modules or oddities that will make your life a lot easier. There is the CPAN module Parse::RecDescent, the standard module Text::Balanced, the byacc program, and Mark-Jason Dominus's excellent *py* tool at http://www.plover.com/~mjd/perl/py/ .

One simple destructive, inside-out approach that you might try is to pull out the smallest nesting parts one at a time:

```
while (s//BEGIN((?:(?!BEGIN)(?!END).)*)END/gs) {
    # do something with $1
}
```

### How do I reverse a string?

Use *reverse()* in scalar context, as documented in the `reverse` entry in the *perlfunc* manpage.

```
$reversed = reverse $string;
```

### How do I expand tabs in a string?

You can do it yourself:

```
1 while $string =~ s/\t+/' ' x (length($&) * 8 - length($`) % 8)/e;
```

Or you can just use the Text::Tabs module (part of the standard perl distribution).

```
use Text::Tabs;
@expanded_lines = expand(@lines_with_tabs);
```

### How do I reformat a paragraph?

Use Text::Wrap (part of the standard perl distribution):

```
use Text::Wrap;
print wrap("\t", '  ', @paragraphs);
```

The paragraphs you give to Text::Wrap should not contain embedded newlines. Text::Wrap doesn't justify the lines (flush-right).

### How can I access/change the first N letters of a string?

There are many ways. If you just want to grab a copy, use *substr()*:

```
$first_byte = substr($a, 0, 1);
```

If you want to modify part of a string, the simplest way is often to use *substr()* as an lvalue:

```
substr($a, 0, 3) = "Tom";
```

Although those with a pattern matching kind of thought process will likely prefer:

```
$a =~ s/^.../Tom/;
```

### How do I change the Nth occurrence of something?

You have to keep track of N yourself. For example, let's say you want to change the fifth occurrence of `"whoever"` or `"whomever"` into `"whosoever"` or `"whomsoever"`, case insensitively.

```
$count = 0;
s{((whom?)ever)}{
    ++$count == 5           # is it the 5th?
        ? "${2}soever"      # yes, swap
        : $1                # renege and leave it there
}igex;
```

In the more general case, you can use the `/g` modifier in a `while` loop, keeping count of matches.

```
$WANT = 3;
$count = 0;
while (/(\w+)\s+fish\b/gi) {
    if (++$count == $WANT) {
        print "The third fish is a $1 one.\n";
        # Warning: don't `last' out of this loop
    }
}
```

That prints out: `"The third fish is a red one."` You can also use a repetition count and repeated pattern like this:

```
/(?:\w+\s+fish\s+){2}(\w+)\s+fish/i;
```

**How can I count the number of occurrences of a substring within a string?**

There are a number of ways, with varying efficiency: If you want a count of a certain single character (X) within a string, you can use the `tr///` function like so:

```
$string = "ThisXlineXhasXsomeXx'sXinXit":
$count = ($string =~ tr/X//);
print "There are $count X charcters in the string";
```

This is fine if you are just looking for a single character. However, if you are trying to count multiple character substrings within a larger string, `tr///` won't work. What you can do is wrap a *while()* loop around a global pattern match. For example, let's count negative integers:

```
$string = "-9 55 48 -2 23 -76 4 14 -44";
while ($string =~ /-\d+/g) { $count++ }
print "There are $count negative numbers in the string";
```

**How do I capitalize all the words on one line?**

To make the first letter of each word upper case:

```
    $line =~ s/\b(\w)/\U$1/g;
```

This has the strange effect of turning "`don't do it`" into "`Don'T Do It`". Sometimes you might want this, instead (Suggested by Brian Foy):

```
$string =~ s/ (
             (^\w)    #at the beginning of the line
               |      # or
             (\s\w)   #preceded by whitespace
               )
            /\U$1/xg;
$string =~ /([\w']+)/\u\L$1/g;
```

To make the whole line upper case:

```
    $line = uc($line);
```

To force each word to be lower case, with the first letter upper case:

```
    $line =~ s/(\w+)/\u\L$1/g;
```

You can (and probably should) enable locale awareness of those characters by placing a `use locale` pragma in your program. See the *perllocale* manpage for endless details on locales.

**How can I split a [character] delimited string except when inside [character]? (Comma-separated files)**

Take the example case of trying to split a string that is comma-separated into its different fields. (We'll pretend you said comma-separated, not comma-delimited, which is different and almost never what you mean.) You can't use `split(/,/)` because you shouldn't split if the comma is inside quotes. For example, take a data line like this:

```
SAR001,"","Cimetrix, Inc","Bob Smith","CAM",N,8,1,0,7,"Error, Core Dumped"
```

Due to the restriction of the quotes, this is a fairly complex problem. Thankfully, we have Jeffrey Friedl, author of a highly recommended book on regular expressions, to handle these for us. He suggests (assuming your string is contained in `$text`):

```
 @new = ();
 push(@new, $+) while $text =~ m{
     "([^\"\\]*(?:\\.[^\"\\]*)*)",?  # groups the phrase inside the quotes
   | ([^,]+),?
   | ,
 }gx;
 push(@new, undef) if substr($text,-1,1) eq ',';
```

If you want to represent quotation marks inside a quotation-mark-delimited field, escape them with backslashes (eg, `"like \"this\""`. Unescaping them is a task addressed earlier in this section.

Alternatively, the Text::ParseWords module (part of the standard perl distribution) lets you say:

```
use Text::ParseWords;
@new = quotewords(",", 0, $text);
```

**How do I strip blank space from the beginning/end of a string?**

Although the simplest approach would seem to be:

```
$string =~ s/^\s*(.*?)\s*$/$1/;
```

This is unneccesarily slow, destructive, and fails with embedded newlines. It is much better faster to do this in two steps:

```
$string =~ s/^\s+//;
$string =~ s/\s+$//;
```

Or more nicely written as:

```
for ($string) {
    s/^\s+//;
    s/\s+$//;
}
```

This idiom takes advantage of the `for(each)` loop's aliasing behavior to factor out common code. You can do this on several strings at once, or arrays, or even the values of a hash if you use a slide:

```
# trim whitespace in the scalar, the array,
# and all the values in the hash
foreach ($scalar, @array, @hash{keys %hash}) {
    s/^\s+//;
    s/\s+$//;
}
```

**How do I extract selected columns from a string?**

Use *substr()* or *unpack()*, both documented in the *perlfunc* manpage. If you prefer thinking in terms of columns instead of widths, you can use this kind of thing:

```
# determine the unpack format needed to split Linux ps output
# arguments are cut columns
my $fmt = cut2fmt(8, 14, 20, 26, 30, 34, 41, 47, 59, 63, 67, 72);

sub cut2fmt {
    my(@positions) = @_;
    my $template  = '';
    my $lastpos   = 1;
    for my $place (@positions) {
        $template .= "A" . ($place - $lastpos) . " ";
        $lastpos   = $place;
    }
    $template .= "A*";
    return $template;
}
```

**How do I find the soundex value of a string?**

Use the standard Text::Soundex module distributed with perl.

**How can I expand variables in text strings?**

Let's assume that you have a string like:

```
$text = 'this has a $foo in it and a $bar';
```

If those were both global variables, then this would suffice:

```
$text =~ s/\$(\w+)/${$1}/g;
```

But since they are probably lexicals, or at least, they could be, you'd have to do this:

```
$text =~ s/(\$\w+)/$1/eeg;
die if $@;                  # needed on /ee, not /e
```

It's probably better in the general case to treat those variables as entries in some special hash. For example:

```
%user_defs = (
    foo  => 23,
    bar  => 19,
);
$text =~ s/\$(\w+)/$user_defs{$1}/g;
```

See also "How do I expand function calls in a string?" in this section of the FAQ.

**What's wrong with always quoting "$vars"?**

The problem is that those double-quotes force stringification, coercing numbers and references into strings, even when you don't want them to be.

If you get used to writing odd things like these:

```
print "$var";       # BAD
$new = "$old";      # BAD
somefunc("$var");   # BAD
```

You'll be in trouble. Those should (in 99.8% of the cases) be the simpler and more direct:

```
print $var;
$new = $old;
somefunc($var);
```

Otherwise, besides slowing you down, you're going to break code when the thing in the scalar is actually neither a string nor a number, but a reference:

```
func(\@array);
sub func {
    my $aref = shift;
    my $oref = "$aref";  # WRONG
}
```

You can also get into subtle problems on those few operations in Perl that actually do care about the difference between a string and a number, such as the magical ++ autoincrement operator or the *syscall()* function.

Stringification also destroys arrays.

```
@lines = `command`;
print "@lines";              # WRONG - extra blanks
print @lines;                # right
```

**Why don't my <<HERE documents work?**

Check for these three things:

1. There must be no space after the << part.

2. There (probably) should be a semicolon at the end.

3. You can't (easily) have any space in front of the tag.

If you want to indent the text in the here document, you can do this:

```
# all in one
($VAR = <<HERE_TARGET) =~ s/^\s+//gm;
    your text
    goes here
HERE_TARGET
```

But the HERE_TARGET must still be flush against the margin. If you want that indented also, you'll have to quote in the indentation.

```
($quote = <<'    FINIS') =~ s/^\s+//gm;
        ...we will have peace, when you and all your works have
        perished--and the works of your dark master to whom you
        would deliver us. You are a liar, Saruman, and a corrupter
        of men's hearts.  --Theoden in /usr/src/perl/taint.c
    FINIS
$quote =~ s/\s*--/\n--/;
```

A nice general-purpose fixer-upper function for indented here documents follows. It expects to be called with a here document as its argument. It looks to see whether each line begins with a common substring, and if so, strips that off. Otherwise, it takes the amount of leading white space found on the first line and removes that much off each subsequent line.

```
sub fix {
    local $_ = shift;
    my ($white, $leader);  # common white space and common leading string
    if (/^\s*(?:([^\w\s]+)(\s*).*\n)(?:\s*\1\2?.*\n)+$/) {
        ($white, $leader) = ($2, quotemeta($1));
    } else {
        ($white, $leader) = (/^(\s+)/, '');
    }
    s/^\s*?$leader(?:$white)?//gm;
    return $_;
}
```

This works with leading special strings, dynamically determined:

```
$remember_the_main = fix<<'    MAIN_INTERPRETER_LOOP';
    @@@ int
    @@@ runops() {
    @@@     SAVEI32(runlevel);
    @@@     runlevel++;
    @@@     while ( op = (*op->op_ppaddr)() ) ;
    @@@     TAINT_NOT;
    @@@     return 0;
    @@@ }
MAIN_INTERPRETER_LOOP
```

Or with a fixed amount of leading white space, with remaining indentation correctly preserved:

```
$poem = fix<<EVER_ON_AND_ON;
   Now far ahead the Road has gone,
      And I must follow, if I can,
   Pursuing it with eager feet,
      Until it joins some larger way
   Where many paths and errands meet.
      And whither then? I cannot say.
            --Bilbo in /usr/src/perl/pp_ctl.c
EVER_ON_AND_ON
```

## Data: Arrays

### What is the difference between `$array[1]` and `@array[1]`?

The former is a scalar value, the latter an array slice, which makes it a list with one (scalar) value. You should use $ when you want a scalar value (most of the time) and @ when you want a list with one scalar value in it (very, very rarely; nearly never, in fact).

Sometimes it doesn't make a difference, but sometimes it does. For example, compare:

```
$good[0] = `some program that outputs several lines`;
```

with

```
@bad[0]  = `same program that outputs several lines`;
```

The **-w** flag will warn you about these matters.

**How can I extract just the unique elements of an array?**

There are several possible ways, depending on whether the array is ordered and whether you wish to preserve the ordering.

a) If @in is sorted, and you want @out to be sorted: (this assumes all true values in the array)

```
$prev = 'nonesuch';
@out = grep($_ ne $prev && ($prev = $_), @in);
```

This is nice in that it doesn't use much extra memory, simulating *uniq* (1)'s behavior of removing only adjacent duplicates. It's less nice in that it won't work with false values like undef, 0, or ""; "0 but true" is ok, though.

b) If you don't know whether @in is sorted:

```
undef %saw;
@out = grep(!$saw{$_}++, @in);
```

c) Like (b), but @in contains only small integers:

```
@out = grep(!$saw[$_]++, @in);
```

d) A way to do (b) without any loops or greps:

```
undef %saw;
@saw{@in} = ();
@out = sort keys %saw;  # remove sort if undesired
```

e) Like (d), but @in contains only small positive integers:

```
undef @ary;
@ary[@in] = @in;
@out = @ary;
```

**How can I tell whether a list or array contains a certain element?**

Hearing the word "in" is an *in*dication that you probably should have used a hash, not a list or array, to store your data. Hashes are designed to answer this question quickly and efficiently. Arrays aren't.

That being said, there are several ways to approach this. If you are going to make this query many times over arbitrary string values, the fastest way is probably to invert the original array and keep an associative array lying about whose keys are the first array's values.

```
@blues = qw/azure cerulean teal turquoise lapis-lazuli/;
undef %is_blue;
for (@blues) { $is_blue{$_} = 1 }
```

Now you can check whether `$is_blue`{$some_color}. It might have been a good idea to keep the blues all in a hash in the first place.

If the values are all small integers, you could use a simple indexed array. This kind of an array will take up less space:

```
@primes = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);
undef @is_tiny_prime;
for (@primes) { $is_tiny_prime[$_] = 1; }
```

Now you check whether `$is_tiny_prime`[$some_number].

If the values in question are integers instead of strings, you can save quite a lot of space by using bit strings instead:

```
@articles = ( 1..10, 150..2000, 2017 );
undef $read;
for (@articles) { vec($read,$_,1) = 1 }
```

Now check whether `vec($read,$n,1)` is true for some `$n`.

Please do not use

```
$is_there = grep $_ eq $whatever, @array;
```

or worse yet

```
$is_there = grep /$whatever/, @array;
```

These are slow (checks every element even if the first matches), inefficient (same reason), and potentially buggy (what if there are regexp characters in `$whatever`?).

**How do I compute the difference of two arrays? How do I compute the intersection of two arrays?**

Use a hash. Here's code to do both and more. It assumes that each element is unique in a given array:

```
@union = @intersection = @difference = ();
%count = ();
foreach $element (@array1, @array2) { $count{$element}++ }
foreach $element (keys %count) {
    push @union, $element;
    push @{ $count{$element} > 1 ? \@intersection : \@difference }, $element;
}
```

**How do I find the first array element for which a condition is true?**

You can use this if you care about the index:

```
for ($i=0; $i < @array; $i++) {
    if ($array[$i] eq "Waldo") {
        $found_index = $i;
        last;
    }
}
```

Now `$found_index` has what you want.

**How do I handle linked lists?**

In general, you usually don't need a linked list in Perl, since with regular arrays, you can push and pop or shift and unshift at either end, or you can use splice to add and/or remove arbitrary number of elements at arbitrary points. Both pop and shift are both $O(1)$ operations on perl's dynamic arrays. In the absence of shifts and pops, push in general needs to reallocate on the order every *log*(N) times, and unshift will need to copy pointers each time.

If you really, really wanted, you could use structures as described in the *perldsc* manpage or the *perltoot* manpage and do just what the algorithm book tells you to do.

**How do I handle circular lists?**

Circular lists could be handled in the traditional fashion with linked lists, or you could just do something like this with an array:

```
unshift(@array, pop(@array));  # the last shall be first
push(@array, shift(@array));   # and vice versa
```

**How do I shuffle an array randomly?**

Use this:

```
# fisher_yates_shuffle( \@array ) :
# generate a random permutation of @array in place
sub fisher_yates_shuffle {
    my $array = shift;
    my $i;
    for ($i = @$array; --$i; ) {
        my $j = int rand ($i+1);
        next if $i == $j;
        @$array[$i,$j] = @$array[$j,$i];
    }
}

fisher_yates_shuffle( \@array );    # permutes @array in place
```

You've probably seen shuffling algorithms that works using splice, randomly picking another element to swap the current element with:

```
srand;
@new = ();
@old = 1 .. 10;  # just a demo
while (@old) {
    push(@new, splice(@old, rand @old, 1));
}
```

This is bad because splice is already *O* (N), and since you do it N times, you just invented a quadratic algorithm; that is, *O* (N**2). This does not scale, although Perl is so efficient that you probably won't notice this until you have rather largish arrays.

**How do I process/modify each element of an array?**

Use `for/foreach`:

```
for (@lines) {
    s/foo/bar/;     # change that word
    y/XZ/ZX/;       # swap those letters
}
```

Here's another; let's compute spherical volumes:

```
for (@volumes = @radii) {   # @volumes has changed parts
    $_ **= 3;
    $_ *= (4/3) * 3.14159;  # this will be constant folded
}
```

If you want to do the same thing to modify the values of the hash, you may not use the `values` function, oddly enough. You need a slice:

```
for $orbit ( @orbits{keys %orbits} ) {
    ($orbit **= 3) *= (4/3) * 3.14159;
}
```

**How do I select a random element from an array?**

Use the *rand()* function (see the `rand` entry in the *perlfunc* manpage):

```
# at the top of the program:
srand;                      # not needed for 5.004 and later

# then later on
$index   = rand @array;
$element = $array[$index];
```

Make sure you *only call srand once per program, if then*. If you are calling it more than once (such as before each call to rand), you're almost certainly doing something wrong.

**How do I permute N elements of a list?**

Here's a little program that generates all permutations of all the words on each line of input. The algorithm embodied in the *permute()* function should work on any list:

```
#!/usr/bin/perl -n
# tsc-permute: permute each word of input
permute([split], []);
sub permute {
    my @items = @{ $_[0] };
    my @perms = @{ $_[1] };
    unless (@items) {
        print "@perms\n";
    } else {
        my(@newitems,@newperms,$i);
        foreach $i (0 .. $#items) {
            @newitems = @items;
            @newperms = @perms;
            unshift(@newperms, splice(@newitems, $i, 1));
            permute([@newitems], [@newperms]);
        }
    }
}
```

**How do I sort an array by (anything)?**

Supply a comparison function to *sort()* (described in the `sort` entry in the *perlfunc* manpage):

```
@list = sort { $a <=> $b } @list;
```

The default sort function is cmp, string comparison, which would sort `(1, 2, 10)` into `(1, 10, 2)`. <=>, used above, is the numerical comparison operator.

If you have a complicated function needed to pull out the part you want to sort on, then don't do it inside the sort function. Pull it out first, because the sort BLOCK can be called many times for the same element. Here's an example of how to pull out the first word after the first number on each item, and then sort those words case-insensitively.

```
@idx = ();
for (@data) {
    ($item) = /\d+\s*(\S+)/;
    push @idx, uc($item);
}
@sorted = @data[ sort { $idx[$a] cmp $idx[$b] } 0 .. $#idx ];
```

Which could also be written this way, using a trick that's come to be known as the Schwartzian Transform:

```
@sorted = map  { $_->[0] }
          sort { $a->[1] cmp $b->[1] }
          map  { [ $_, uc((/\d+\s*(\S+)/ )[0] ] } @data;
```

If you need to sort on several fields, the following paradigm is useful.

```
@sorted = sort { field1($a) <=> field1($b) ||
                 field2($a) cmp field2($b) ||
                 field3($a) cmp field3($b)
               }     @data;
```

This can be conveniently combined with precalculation of keys as given above.

See http://www.perl.com/CPAN/doc/FMTEYEWTK/sort.html for more about this approach.

See also the question below on sorting hashes.

**How do I manipulate arrays of bits?**

Use *pack()* and *unpack()*, or else *vec()* and the bitwise operations.

For example, this sets `$vec` to have bit N set if `$ints`[N] was set:

```
$vec = '';
foreach(@ints) { vec($vec,$_,1) = 1 }
```

And here's how, given a vector in `$vec`, you can get those bits into your `@ints` array:

```
sub bitvec_to_list {
    my $vec = shift;
    my @ints;
    # Find null-byte density then select best algorithm
    if ($vec =~ tr/\0// / length $vec > 0.95) {
        use integer;
        my $i;
        # This method is faster with mostly null-bytes
        while($vec =~ /[^\0]/g ) {
            $i = -9 + 8 * pos $vec;
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
        }
    } else {
        # This method is a fast general algorithm
        use integer;
        my $bits = unpack "b*", $vec;
        push @ints, 0 if $bits =~ s/^(\d)// && $1;
        push @ints, pos $bits while($bits =~ /1/g);
    }
    return \@ints;
}
```

This method gets faster the more sparse the bit vector is. (Courtesy of Tim Bunce and Winfried Koenig.)

### Why does *defined()* return true on empty arrays and hashes?

See the `defined` entry in the *perlfunc* manpage in the 5.004 release or later of Perl.

## Data: Hashes (Associative Arrays)

### How do I process an entire hash?

Use the *each()* function (see the `each` entry in the *perlfunc* manpage) if you don't care whether it's sorted:

```
while ( ($key, $value) = each %hash) {
    print "$key = $value\n";
}
```

If you want it sorted, you'll have to use *foreach()* on the result of sorting the keys as shown in an earlier question.

### What happens if I add or remove keys from a hash while iterating over it?

Don't do that.

### How do I look up a hash element by value?

Create a reverse hash:

```
%by_value = reverse %by_key;
$key = $by_value{$value};
```

That's not particularly efficient. It would be more space-efficient to use:

```
while (($key, $value) = each %by_key) {
    $by_value{$value} = $key;
}
```

If your hash could have repeated values, the methods above will only find one of the associated keys. This may or may not worry you.

**How can I know how many entries are in a hash?**

If you mean how many keys, then all you have to do is take the scalar sense of the *keys()* function:

```
$num_keys = scalar keys %hash;
```

In void context it just resets the iterator, which is faster for tied hashes.

**How do I sort a hash (optionally by value instead of key)?**

Internally, hashes are stored in a way that prevents you from imposing an order on key-value pairs. Instead, you have to sort a list of the keys or values:

```
@keys = sort keys %hash;     # sorted by key
@keys = sort {
                $hash{$a} cmp $hash{$b}
        } keys %hash;        # and by value
```

Here we'll do a reverse numeric sort by value, and if two keys are identical, sort by length of key, and if that fails, by straight ASCII comparison of the keys (well, possibly modified by your locale -- see the *perllocale* manpage).

```
@keys = sort {
            $hash{$b} <=> $hash{$a}
                      ||
            length($b) <=> length($a)
                      ||
                  $a cmp $b
} keys %hash;
```

**How can I always keep my hash sorted?**

You can look into using the DB_File module and *tie()* using the `$DB_BTREE` hash bindings as documented in the section on *In Memory Databases* in the *DB_File* manpage. The Tie::IxHash module from CPAN might also be instructive.

**What's the difference between "delete" and "undef" with hashes?**

Hashes are pairs of scalars: the first is the key, the second is the value. The key will be coerced to a string, although the value can be any kind of scalar: string, number, or reference. If a key `$key` is present in the array, `exists($key)` will return true. The value for a given key can be `undef`, in which case `$array{$key}` will be `undef` while `$exists{$key}` will return true. This corresponds to (`$key`, `undef`) being in the hash.

Pictures help... here's the `%ary` table:

```
  keys  values
+------+------+
|  a   |  3   |
|  x   |  7   |
|  d   |  0   |
|  e   |  2   |
+------+------+
```

And these conditions hold

```
$ary{'a'}                       is true
$ary{'d'}                       is false
defined $ary{'d'}               is true
defined $ary{'a'}               is true
exists $ary{'a'}                is true (perl5 only)
grep ($_ eq 'a', keys %ary)     is true
```

If you now say

```
undef $ary{'a'}
```

your table now reads:

```
  keys  values
+------+------+
|  a   | undef|
|  x   |  7   |
|  d   |  0   |
|  e   |  2   |
+------+------+
```

and these conditions now hold; changes in caps:

```
$ary{'a'}                       is FALSE
$ary{'d'}                       is false
defined $ary{'d'}               is true
defined $ary{'a'}               is FALSE
exists $ary{'a'}                is true (perl5 only)
grep ($_ eq 'a', keys %ary)     is true
```

Notice the last two: you have an undef value, but a defined key!

Now, consider this:

```
delete $ary{'a'}
```

your table now reads:

```
  keys  values
+------+------+
|  x   |  7   |
|  d   |  0   |
|  e   |  2   |
+------+------+
```

and these conditions now hold; changes in caps:

```
$ary{'a'}                       is false
$ary{'d'}                       is false
defined $ary{'d'}               is true
defined $ary{'a'}               is false
exists $ary{'a'}                is FALSE (perl5 only)
grep ($_ eq 'a', keys %ary)     is FALSE
```

See, the whole entry is gone!

**Why don't my tied hashes make the defined/exists distinction?**

They may or may not implement the *EXISTS()* and *DEFINED()* methods differently. For example, there isn't the concept of undef with hashes that are tied to DBM* files. This means the true/false tables above will give different results when used on such a hash. It also means that exists and defined do the same thing with a DBM* file, and what they end up doing is not what they do with ordinary hashes.

**How do I reset an** *each()* operation part-way through?

Using `keys %hash` in scalar context returns the number of keys in the hash *and* resets the iterator associated with the hash. You may need to do this if you use `last` to exit a loop early so that when you re-enter it, the hash iterator has been reset.

**How can I get the unique keys from two hashes?**

First you extract the keys from the hashes into arrays, and then solve the uniquifying the array problem described above. For example:

```
%seen = ();
for $element (keys(%foo), keys(%bar)) {
    $seen{$element}++;
}
@uniq = keys %seen;
```

Or more succinctly:

```
@uniq = keys %{{%foo,%bar}};
```

Or if you really want to save space:

```
%seen = ();
while (defined ($key = each %foo)) {
    $seen{$key}++;
}
while (defined ($key = each %bar)) {
    $seen{$key}++;
}
@uniq = keys %seen;
```

**How can I store a multidimensional array in a DBM file?**

Either stringify the structure yourself (no fun), or else get the MLDBM (which uses Data::Dumper) module from CPAN and layer it on top of either DB_File or GDBM_File.

**How can I make my hash remember the order I put elements into it?**

Use the Tie::IxHash from CPAN.

```
use Tie::IxHash;
tie(%myhash, Tie::IxHash);
for ($i=0; $i<20; $i++) {
    $myhash{$i} = 2*$i;
}
@keys = keys %myhash;
# @keys = (0,1,2,3,...)
```

### Why does passing a subroutine an undefined element in a hash create it?

If you say something like:

```
somefunc($hash{"nonesuch key here"});
```

Then that element ``autovivifies''; that is, it springs into existence whether you store something there or not. That's because functions get scalars passed in by reference. If *somefunc()* modifies `$_[0]`, it has to be ready to write it back into the caller's version.

This has been fixed as of perl5.004.

Normally, merely accessing a key's value for a nonexistent key does *not* cause that key to be forever there. This is different than awk's behavior.

### How can I make the Perl equivalent of a C structure/C++ class/hash or array of hashes or arrays?

Use references (documented in the *perlref* manpage). Examples of complex data structures are given in the *perldsc* manpage and the *perllol* manpage. Examples of structures and object-oriented classes are in the *perltoot* manpage.

### How can I use a reference as a hash key?

You can't do this directly, but you could use the standard Tie::Refhash module distributed with perl.

## Data: Misc

### How do I handle binary data correctly?

Perl is binary clean, so this shouldn't be a problem. For example, this works fine (assuming the files are found):

```
if (`cat /vmunix` =~ /gzip/) {
    print "Your kernel is GNU-zip enabled!\n";
}
```

On some systems, however, you have to play tedious games with ``text'' versus ``binary'' files. See the section on *binmode* in the *perlfunc* manpage.

If you're concerned about 8-bit ASCII data, then see the *perllocale* manpage.

If you want to deal with multibyte characters, however, there are some gotchas. See the section on Regular Expressions.

### How do I determine whether a scalar is a number/whole/integer/float?

Assuming that you don't care about IEEE notations like ``NaN'' or ``Infinity'', you probably just want to use a regular expression.

```
warn "has nondigits"        if     /\D/;
 warn "not a natural number" unless /^\d+$/;             # rejects -3
 warn "not an integer"       unless /^-?\d+$/;           # rejects +3
warn "not an integer"       unless /^[+-]?\d+$/;
warn "not a decimal number" unless /^-?\d+\.?\d*$/;  # rejects .2
warn "not a decimal number" unless /^-?(?:\d+(?:\.\d*)?|\.\d+)$/;
warn "not a C float"
    unless /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/;
```

If you're on a POSIX system, Perl's supports the `POSIX::strtod` function. Its semantics are somewhat cumbersome, so here's a `getnum` wrapper function for more convenient access. This function takes a string and returns the number it found, or `undef` for input that isn't a C float. The `is_numeric` function is a front end to `getnum` if you just want to say, "Is this a float?"

```
sub getnum {
    use POSIX qw(strtod);
    my $str = shift;
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;
    $! = 0;
    my($num, $unparsed) = strtod($str);
    if (($str eq '') || ($unparsed != 0) || $!) {
        return undef;
    } else {
        return $num;
    }
}

sub is_numeric { defined &getnum }
```

Or you could check out http://www.perl.com/CPAN/modules/by-module/String/String-Scanf-1.1.tar.gz instead. The POSIX module (part of the standard Perl distribution) provides the `strtol` and `strtod` for converting strings to double and longs, respectively.

**How do I keep persistent data across program calls?**

For some specific applications, you can use one of the DBM modules. See the *AnyDBM_File* manpage. More generically, you should consult the FreezeThaw, Storable, or Class::Eroot modules from CPAN.

**How do I print out or copy a recursive data structure?**

The Data::Dumper module on CPAN is nice for printing out data structures, and FreezeThaw for copying them. For example:

```
use FreezeThaw qw(freeze thaw);
$new = thaw freeze $old;
```

Where `$old` can be (a reference to) any kind of data structure you'd like. It will be deeply copied.

**How do I define methods for every class/object?**

Use the UNIVERSAL class (see the *UNIVERSAL* manpage).

### How do I verify a credit card checksum?

Get the Business::CreditCard module from CPAN.

## AUTHOR AND COPYRIGHT

**NAME**

perlfaq5 − Files and Formats ($Revision: 1.24 $, `$Date:` 1998/07/05 15:07:20 $)

**DESCRIPTION**

This section deals with I/O and the "f" issues: filehandles, flushing, formats, and footers.

**How do I flush/unbuffer an output filehandle? Why must I do this?**

The C standard I/O library (stdio) normally buffers characters sent to devices. This is done for efficiency reasons, so that there isn't a system call for each byte. Any time you use *print()* or *write()* in Perl, you go though this buffering. *syswrite()* circumvents stdio and buffering.

In most stdio implementations, the type of output buffering and the size of the buffer varies according to the type of device. Disk files are block buffered, often with a buffer size of more than 2k. Pipes and sockets are often buffered with a buffer size between 1/2 and 2k. Serial devices (e.g. modems, terminals) are normally line-buffered, and stdio sends the entire line when it gets the newline.

Perl does not support truly unbuffered output (except insofar as you can `syswrite(OUT, $char, 1)`). What it does instead support is "command buffering", in which a physical write is performed after every output command. This isn't as hard on your system as unbuffering, but does get the output where you want it when you want it.

If you expect characters to get to your device when you print them there, you'll want to autoflush its handle. Use *select()* and the `$|` variable to control autoflushing (see the section on *$|* in the *perlvar* manpage and the `select` entry in the *perlfunc* manpage):

```
$old_fh = select(OUTPUT_HANDLE);
$| = 1;
select($old_fh);
```

Or using the traditional idiom:

```
select((select(OUTPUT_HANDLE), $| = 1)[0]);
```

Or if don't mind slowly loading several thousand lines of module code just because you're afraid of the `$|` variable:

```
use FileHandle;
open(DEV, "+</dev/tty");      # ceci n'est pas une pipe
DEV->autoflush(1);
```

or the newer IO::* modules:

```
use IO::Handle;
open(DEV, ">/dev/printer");   # but is this?
DEV->autoflush(1);
```

or even this:

```
use IO::Socket;               # this one is kinda a pipe?
$sock = IO::Socket::INET->new(PeerAddr => 'www.perl.com',
                              PeerPort => 'http(80)',
                              Proto    => 'tcp');
die "$!" unless $sock;

$sock->autoflush();
print $sock "GET / HTTP/1.0" . "\015\012" x 2;
$document = join('', <$sock>);
print "DOC IS: $document\n";
```

Note the bizarrely hardcoded carriage return and newline in their octal equivalents. This is the ONLY way (currently) to assure a proper flush on all platforms, including Macintosh. That the way things work in network programming: you really should specify the exact bit pattern on the network line terminator. In practice, `"\n\n"` often works, but this is not portable.

See the *perlfaq9* manpage for other examples of fetching URLs over the web.

**How do I change one line in a file/delete a line in a file/insert a line in the middle of a file/append to the beginning of a file?**

Although humans have an easy time thinking of a text file as being a sequence of lines that operates much like a stack of playing cards -- or punch cards -- computers usually see the text file as a sequence of bytes. In general, there's no direct way for Perl to seek to a particular line of a file, insert text into a file, or remove text from a file.

(There are exceptions in special circumstances. You can add or remove at the very end of the file. Another is replacing a sequence of bytes with another sequence of the same length. Another is using the `$DB_RECNO` array bindings as documented in the *DB_File* manpage. Yet another is manipulating files with all lines the same length.)

The general solution is to create a temporary copy of the text file with the changes you want, then copy that over the original. This assumes no locking.

```
$old = $file;
$new = "$file.tmp.$$";
$bak = "$file.bak";

open(OLD, "< $old")         or die "can't open $old: $!";
open(NEW, "> $new")         or die "can't open $new: $!";

# Correct typos, preserving case
while (<OLD>) {
    s/\b(p)earl\b/${1}erl/i;
    (print NEW $_)          or die "can't write to $new: $!";
}

close(OLD)                  or die "can't close $old: $!";
close(NEW)                  or die "can't close $new: $!";

rename($old, $bak)          or die "can't rename $old to $bak: $!";
rename($new, $old)          or die "can't rename $new to $old: $!";
```

Perl can do this sort of thing for you automatically with the `-i` command-line switch or the closely-related `$^I` variable (see the *perlrun* manpage for more details). Note that `-i` may require a suffix on some non-Unix systems; see the platform-specific documentation that came with your port.

```
# Renumber a series of tests from the command line
perl -pi -e 's/(^\s+test\s+)\d+/ $1 . ++$count /e' t/op/taint.t
```

```
# form a script
local($^I, @ARGV) = ('.bak', glob("*.c"));
while (<>) {
    if ($. == 1) {
        print "This line should appear at the top of each file\n";
    }
    s/\b(p)earl\b/${1}erl/i;        # Correct typos, preserving case
    print;
    close ARGV if eof;              # Reset $.
}
```

If you need to seek to an arbitrary line of a file that changes infrequently, you could build up an index of byte positions of where the line ends are in the file. If the file is large, an index of every tenth or hundredth line end would allow you to seek and read fairly efficiently. If the file is sorted, try the look.pl library (part of the standard perl distribution).

In the unique case of deleting lines at the end of a file, you can use *tell()* and *truncate()*. The following code snippet deletes the last line of a file without making a copy or reading the whole file into memory:

```
open (FH, "+< $file");
while ( <FH> ) { $addr = tell(FH) unless eof(FH) }
truncate(FH, $addr);
```

Error checking is left as an exercise for the reader.

### How do I count the number of lines in a file?

One fairly efficient way is to count newlines in the file. The following program uses a feature of tr///, as documented in the *perlop* manpage. If your text file doesn't end with a newline, then it's not really a proper text file, so this may report one fewer line than you expect.

```
$lines = 0;
open(FILE, $filename) or die "Can't open `$filename': $!";
while (sysread FILE, $buffer, 4096) {
    $lines += ($buffer =~ tr/\n//);
}
close FILE;
```

This assumes no funny games with newline translations.

### How do I make a temporary file name?

Use the `new_tmpfile` class method from the IO::File module to get a filehandle opened for reading and writing. Use this if you don't need to know the file's name.

```
    use IO::File;
$fh = IO::File->new_tmpfile()
        or die "Unable to make new temporary file: $!";
```

Or you can use the `tmpnam` function from the POSIX module to get a filename that you then open yourself. Use this if you do need to know the file's name.

```
use Fcntl;
use POSIX qw(tmpnam);
```

```
# try new temporary filenames until we get one that didn't already
# exist;  the check should be unnecessary, but you can't be too careful
do { $name = tmpnam() }
    until sysopen(FH, $name, O_RDWR|O_CREAT|O_EXCL);

# install atexit-style handler so that when we exit or die,
# we automatically delete this temporary file
END { unlink($name) or die "Couldn't unlink $name : $!" }

# now go on to use the file ...
```

If you're committed to doing this by hand, use the process ID and/or the current time-value. If you need to have many temporary files in one process, use a counter:

```
BEGIN {
    use Fcntl;
    my $temp_dir = -d '/tmp' ? '/tmp' : $ENV{TMP} || $ENV{TEMP};
    my $base_name = sprintf("%s/%d-%d-0000", $temp_dir, $$, time());
    sub temp_file {
        local *FH;
        my $count = 0;
        until (defined(fileno(FH)) || $count++ > 100) {
            $base_name =~ s/-(\d+)$/"-" . (1 + $1)/e;
            sysopen(FH, $base_name, O_WRONLY|O_EXCL|O_CREAT);
        }
        if (defined(fileno(FH))
            return (*FH, $base_name);
        } else {
            return ();
        }
    }
}
```

**How can I manipulate fixed-record-length files?**

The most efficient way is using *pack()* and *unpack()*. This is faster than using *substr()* when take many, many strings. It is slower for just a few.

Here is a sample chunk of code to break up and put back together again some fixed-format input lines, in this case from the output of a normal, Berkeley-style ps:

```
# sample input line:
#   15158 p5  T      0:00 perl /home/tchrist/scripts/now-what
$PS_T = 'A6 A4 A7 A5 A*';
open(PS, "ps|");
print scalar <PS>;
while (<PS>) {
    ($pid, $tt, $stat, $time, $command) = unpack($PS_T, $_);
    for $var (qw!pid tt stat time command!) {
        print "$var: <$$var>\n";
    }
    print 'line=', pack($PS_T, $pid, $tt, $stat, $time, $command),
            "\n";
}
```

We've used `$$var` in a way that forbidden by `use strict 'refs'`. That is, we've promoted a string to a scalar variable reference using symbolic references. This is ok in small programs, but doesn't scale well. It also only works on global variables, not lexicals.

**How can I make a filehandle local to a subroutine? How do I pass filehandles between subroutines? How do I make an array of filehandles?**

The fastest, simplest, and most direct way is to localize the typeglob of the filehandle in question:

```
local *TmpHandle;
```

Typeglobs are fast (especially compared with the alternatives) and reasonably easy to use, but they also have one subtle drawback. If you had, for example, a function named *TmpHandle()*, or a variable named `%TmpHandle`, you just hid it from yourself.

```
sub findme {
    local *HostFile;
    open(HostFile, "</etc/hosts") or die "no /etc/hosts: $!";
    local $_;               # <- VERY IMPORTANT
    while (<HostFile>) {
        print if /\b127\.(0\.0\.)?1\b/;
    }
    # *HostFile automatically closes/disappears here
}
```

Here's how to use this in a loop to open and store a bunch of filehandles. We'll use as values of the hash an ordered pair to make it easy to sort the hash in insertion order.

```
@names = qw(motd termcap passwd hosts);
my $i = 0;
foreach $filename (@names) {
    local *FH;
    open(FH, "/etc/$filename") || die "$filename: $!";
    $file{$filename} = [ $i++, *FH ];
}

# Using the filehandles in the array
foreach $name (sort { $file{$a}[0] <=> $file{$b}[0] } keys %file) {
    my $fh = $file{$name}[1];
    my $line = <$fh>;
    print "$name $. $line";
}
```

For passing filehandles to functions, the easiest way is to prefer them with a star, as in *func* (*STDIN). See the section on *Passing Filehandles* in the *perlfaq7* manpage for details.

If you want to create many, anonymous handles, you should check out the Symbol, FileHandle, or IO::Handle (etc.) modules. Here's the equivalent code with Symbol::gensym, which is reasonably light-weight:

```
foreach $filename (@names) {
    use Symbol;
    my $fh = gensym();
    open($fh, "/etc/$filename") || die "open /etc/$filename: $!";
    $file{$filename} = [ $i++, $fh ];
}
```

Or here using the semi-object-oriented FileHandle, which certainly isn't light-weight:

```
use FileHandle;
```

```
foreach $filename (@names) {
    my $fh = FileHandle->new("/etc/$filename") or die "$filename: $!";
    $file{$filename} = [ $i++, $fh ];
}
```

Please understand that whether the filehandle happens to be a (probably localized) typeglob or an anonymous handle from one of the modules, in no way affects the bizarre rules for managing indirect handles. See the next question.

### How can I use a filehandle indirectly?

An indirect filehandle is using something other than a symbol in a place that a filehandle is expected. Here are ways to get those:

```
$fh =   SOME_FH;       # bareword is strict-subs hostile
$fh =  "SOME_FH";      # strict-refs hostile; same package only
$fh =  *SOME_FH;       # typeglob
$fh = \*SOME_FH;       # ref to typeglob (bless-able)
$fh =  *SOME_FH{IO};   # blessed IO::Handle from *SOME_FH typeglob
```

Or to use the `new` method from the FileHandle or IO modules to create an anonymous filehandle, store that in a scalar variable, and use it as though it were a normal filehandle.

```
use FileHandle;
$fh = FileHandle->new();

use IO::Handle;                     # 5.004 or higher
$fh = IO::Handle->new();
```

Then use any of those as you would a normal filehandle. Anywhere that Perl is expecting a filehandle, an indirect filehandle may be used instead. An indirect filehandle is just a scalar variable that contains a filehandle. Functions like `print`, `open`, `seek`, or the functions or the `<FH>` diamond operator will accept either a read filehandle or a scalar variable containing one:

```
($ifh, $ofh, $efh) = (*STDIN, *STDOUT, *STDERR);
print $ofh "Type it: ";
$got = <$ifh>
print $efh "What was that: $got";
```

Of you're passing a filehandle to a function, you can write the function in two ways:

```
sub accept_fh {
    my $fh = shift;
    print $fh "Sending to indirect filehandle\n";
}
```

Or it can localize a typeglob and use the filehandle directly:

```
sub accept_fh {
    local *FH = shift;
    print  FH "Sending to localized filehandle\n";
}
```

Both styles work with either objects or typeglobs of real filehandles. (They might also work with strings under some circumstances, but this is risky.)

```
accept_fh(*STDOUT);
accept_fh($handle);
```

In the examples above, we assigned the filehandle to a scalar variable before using it. That is because only simple scalar variables, not expressions or subscripts into hashes or arrays, can be used with built-ins like `print`, `printf`, or the diamond operator. These are illegal and won't even compile:

```
@fd = (*STDIN, *STDOUT, *STDERR);
print $fd[1] "Type it: ";                          # WRONG
$got = <$fd[0]>                                    # WRONG
print $fd[2] "What was that: $got";                # WRONG
```

With `print` and `printf`, you get around this by using a block and an expression where you would place the filehandle:

```
print  { $fd[1] } "funny stuff\n";
printf { $fd[1] } "Pity the poor %x.\n", 3_735_928_559;
# Pity the poor deadbeef.
```

That block is a proper block like any other, so you can put more complicated code there. This sends the message out to one of two places:

```
$ok = -x "/bin/cat";
print { $ok ? $fd[1] : $fd[2] } "cat stat $ok\n";
print { $fd[ 1+ ($ok || 0) ]  } "cat stat $ok\n";
```

This approach of treating `print` and `printf` like object methods calls doesn't work for the diamond operator. That's because it's a real operator, not just a function with a comma-less argument. Assuming you've been storing typeglobs in your structure as we did above, you can use the built-in function named `readline` to reads a record just as <> does. Given the initialization shown above for `@fd`, this would work, but only because *readline()* require a typeglob. It doesn't work with objects or strings, which might be a bug we haven't fixed yet.

```
$got = readline($fd[0]);
```

Let it be noted that the flakiness of indirect filehandles is not related to whether they're strings, typeglobs, objects, or anything else. It's the syntax of the fundamental operators. Playing the object game doesn't help you at all here.

### How can I set up a footer format to be used with *write()*?

There's no builtin way to do this, but the *perlform* manpage has a couple of techniques to make it possible for the intrepid hacker.

### How can I *write()* into a string?

See the *perlform* manpage for an *swrite()* function.

### How can I output my numbers with commas added?

This one will do it for you:

```
sub commify {
    local $_  = shift;
    1 while s/^(-?\d+)(\d{3})/$1,$2/;
    return $_;
}

$n = 23659019423.2331;
print "GOT: ", commify($n), "\n";

GOT: 23,659,019,423.2331
```

You can't just:

```
s/^(-?\d+)(\d{3})/$1,$2/g;
```

because you have to put the comma in and then recalculate your position.

Alternatively, this commifies all numbers in a line regardless of whether they have decimal portions, are preceded by + or –, or whatever:

```
# from Andrew Johnson <ajohnson@gpu.srv.ualberta.ca>
sub commify {
   my $input = shift;
    $input = reverse $input;
    $input =~ s<(\d\d\d)(?=\d)(?!\d*\.)><$1,>g;
    return reverse $input;
}
```

**How can I translate tildes (~) in a filename?**

Use the <> (*glob()*) operator, documented in the *perlfunc* manpage. This requires that you have a shell installed that groks tildes, meaning csh or tcsh or (some versions of) ksh, and thus may have portability problems. The Glob::KGlob module (available from CPAN) gives more portable glob functionality.

Within Perl, you may use this directly:

```
        $filename =~ s{
          ^ ~             # find a leading tilde
          (               # save this in $1
              [^/]        # a non-slash character
                    *     # repeated 0 or more times (0 means me)
          )
        }{
          $1
              ? (getpwnam($1))[7]
              : ( $ENV{HOME} || $ENV{LOGDIR} )
        }ex;
```

**How come when I open a file read-write it wipes it out?**

Because you're using something like this, which truncates the file and *then* gives you read-write access:

```
open(FH, "+> /path/name");           # WRONG (almost always)
```

Whoops. You should instead use this, which will fail if the file doesn't exist. Using ">" always clobbers or creates. Using "<" never does either. The "+" doesn't change this.

Here are examples of many kinds of file opens. Those using *sysopen()* all assume

```
use Fcntl;
```

To open file for reading:

```
open(FH, "< $path")                                 || die $!;
sysopen(FH, $path, O_RDONLY)                        || die $!;
```

To open file for writing, create new file if needed or else truncate old file:

```
open(FH, "> $path") || die $!;
sysopen(FH, $path, O_WRONLY|O_TRUNC|O_CREAT)        || die $!;
sysopen(FH, $path, O_WRONLY|O_TRUNC|O_CREAT, 0666)  || die $!;
```

To open file for writing, create new file, file must not exist:

```
sysopen(FH, $path, O_WRONLY|O_EXCL|O_CREAT)         || die $!;
sysopen(FH, $path, O_WRONLY|O_EXCL|O_CREAT, 0666)   || die $!;
```

To open file for appending, create if necessary:

```
open(FH, ">> $path") || die $!;
sysopen(FH, $path, O_WRONLY|O_APPEND|O_CREAT)       || die $!;
sysopen(FH, $path, O_WRONLY|O_APPEND|O_CREAT, 0666) || die $!;
```

To open file for appending, file must exist:

```
sysopen(FH, $path, O_WRONLY|O_APPEND)               || die $!;
```

To open file for update, file must exist:

```
open(FH, "+< $path")                                || die $!;
sysopen(FH, $path, O_RDWR)                          || die $!;
```

To open file for update, create file if necessary:

```
sysopen(FH, $path, O_RDWR|O_CREAT)                  || die $!;
sysopen(FH, $path, O_RDWR|O_CREAT, 0666)            || die $!;
```

To open file for update, file must not exist:

```
sysopen(FH, $path, O_RDWR|O_EXCL|O_CREAT)           || die $!;
sysopen(FH, $path, O_RDWR|O_EXCL|O_CREAT, 0666)     || die $!;
```

To open a file without blocking, creating if necessary:

```
sysopen(FH, "/tmp/somefile", O_WRONLY|O_NDELAY|O_CREAT)
        or die "can't open /tmp/somefile: $!":
```

Be warned that neither creation nor deletion of files is guaranteed to be an atomic operation over NFS. That is, two processes might both successful create or unlink the same file! Therefore O_EXCL isn't so exclusive as you might wish.

### Why do I sometimes get an "Argument list too long" when I use <*>?

The <> operator performs a globbing operation (see above). By default *glob()* forks *csh*(1) to do the actual glob expansion, but csh can't handle more than 127 items and so gives the error message `Argument list too long`. People who installed tcsh as csh won't have this problem, but their users may be surprised by it.

To get around this, either do the glob yourself with `Dirhandles` and patterns, or use a module like Glob::KGlob, one that doesn't use the shell to do globbing.

### Is there a leak/bug in *glob()*?

Due to the current implementation on some operating systems, when you use the *glob()* function or its angle-bracket alias in a scalar context, you may cause a leak and/or unpredictable behavior. It's best therefore to use *glob()* only in list context.

### How can I open a file with a leading ">" or trailing blanks?

Normally perl ignores trailing blanks in filenames, and interprets certain leading characters (or a trailing "|") to mean something special. To avoid this, you might want to use a routine like this. It makes incomplete pathnames into explicit relative ones, and tacks a trailing null byte on the name to make perl leave it alone:

```
sub safe_filename {
    local $_  = shift;
    return m#^/#
            ? "$_\0"
            : "./$_\0";
}

$fn = safe_filename("<<<something really wicked   ");
open(FH, "> $fn") or "couldn't open $fn: $!";
```

You could also use the *sysopen()* function (see the `sysopen` entry in the *perlfunc* manpage).

### How can I reliably rename a file?

Well, usually you just use Perl's *rename()* function. But that may not work everywhere, in particular, renaming files across file systems. If your operating system supports a *mv*(1) program or its moral equivalent, this works:

```
rename($old, $new) or system("mv", $old, $new);
```

It may be more compelling to use the File::Copy module instead. You just copy to the new file to the new name (checking return values), then delete the old one. This isn't really the same semantics as a real *rename()*, though, which preserves metainformation like permissions, timestamps, inode info, etc.

The newer version of File::Copy export a *move()* function.

### How can I lock a file?

Perl's builtin *flock()* function (see the *perlfunc* manpage for details) will call *flock*(2) if that exists, *fcntl*(2) if it doesn't (on perl version 5.004 and later), and *lockf*(3) if neither of the two previous system calls exists. On some systems, it may even use a different form of native locking. Here are some gotchas with Perl's *flock()*:

1 Produces a fatal error if none of the three system calls (or their close equivalent) exists.

2 *lockf*(3) does not provide shared locking, and requires that the filehandle be open for writing (or appending, or read/writing).

3 Some versions of *flock()* can't lock files over a network (e.g. on NFS file systems), so you'd need to force the use of *fcntl*(2) when you build Perl. See the flock entry of the *perlfunc* manpage, and the *INSTALL* file in the source distribution for information on building Perl to do this.

### What can't I just *open*(FH, ">file.lock")?

A common bit of code **NOT TO USE** is this:

```
sleep(3) while -e "file.lock";      # PLEASE DO NOT USE
open(LCK, "> file.lock");           # THIS BROKEN CODE
```

This is a classic race condition: you take two steps to do something which must be done in one. That's why computer hardware provides an atomic test-and-set instruction. In theory, this "ought" to work:

```
sysopen(FH, "file.lock", O_WRONLY|O_EXCL|O_CREAT)
            or die "can't open  file.lock: $!":
```

except that lamentably, file creation (and deletion) is not atomic over NFS, so this won't work (at least, not every time) over the net. Various schemes involving involving *link()* have been suggested, but these tend to involve busy-wait, which is also subdesirable.

**I still don't get locking. I just want to increment the number in the file. How can I do this?**

Didn't anyone ever tell you web-page hit counters were useless? They don't count number of hits, they're a waste of time, and they serve only to stroke the writer's vanity. Better to pick a random number. It's more realistic.

Anyway, this is what you can do if you can't help yourself.

```
use Fcntl;
sysopen(FH, "numfile", O_RDWR|O_CREAT)       or die "can't open numfile: $!";
flock(FH, 2)                                 or die "can't flock numfile: $!";
$num = <FH> || 0;
seek(FH, 0, 0)                               or die "can't rewind numfile: $!";
truncate(FH, 0)                              or die "can't truncate numfile: $!"
(print FH $num+1, "\n")                      or die "can't write numfile: $!";
# DO NOT UNLOCK THIS UNTIL YOU CLOSE
close FH                                     or die "can't close numfile: $!";
```

Here's a much better web-page hit counter:

```
$hits = int( (time() - 850_000_000) / rand(1_000) );
```

If the count doesn't impress your friends, then the code might. :–)

**How do I randomly update a binary file?**

If you're just trying to patch a binary, in many cases something as simple as this works:

```
perl -i -pe 's{window manager}{window mangler}g' /usr/bin/emacs
```

However, if you have fixed sized records, then you might do something more like this:

```
$RECSIZE = 220; # size of record, in bytes
$recno   = 37;  # which record to update
open(FH, "+<somewhere") || die "can't update somewhere: $!";
seek(FH, $recno * $RECSIZE, 0);
read(FH, $record, $RECSIZE) == $RECSIZE || die "can't read record $recno: $!";
# munge the record
seek(FH, $recno * $RECSIZE, 0);
print FH $record;
close FH;
```

Locking and error checking are left as an exercise for the reader. Don't forget them, or you'll be quite sorry.

**How do I get a file's timestamp in perl?**

If you want to retrieve the time at which the file was last read, written, or had its meta-data (owner, etc) changed, you use the **–M**, **–A**, or **–C** filetest operations as documented in the *perlfunc* manpage. These retrieve the age of the file (measured against the start-time of your program) in days as a floating point number. To retrieve the "raw" time in seconds since the epoch, you would call the stat function, then use *localtime()*, *gmtime()*, or *POSIX::strftime()* to convert this into human-readable form.

Here's an example:

```
$write_secs = (stat($file))[9];
printf "file %s updated at %s\n", $file,
    scalar localtime($write_secs);
```

If you prefer something more legible, use the File::stat module (part of the standard distribution in version 5.004 and later):

```
use File::stat;
use Time::localtime;
$date_string = ctime(stat($file)->mtime);
print "file $file updated at $date_string\n";
```

Error checking is left as an exercise for the reader.

### How do I set a file's timestamp in perl?

You use the *utime()* function documented in the `utime` entry in the *perlfunc* manpage. By way of example, here's a little program that copies the read and write times from its first argument to all the rest of them.

```
if (@ARGV < 2) {
    die "usage: cptimes timestamp_file other_files ...\n";
}
$timestamp = shift;
($atime, $mtime) = (stat($timestamp))[8,9];
utime $atime, $mtime, @ARGV;
```

Error checking is left as an exercise for the reader.

Note that *utime()* currently doesn't work correctly with Win95/NT ports. A bug has been reported. Check it carefully before using it on those platforms.

### How do I print to more than one file at once?

If you only have to do this once, you can do this:

```
for $fh (FH1, FH2, FH3) { print $fh "whatever\n" }
```

To connect up to one filehandle to several output filehandles, it's easiest to use the *tee*(1) program if you have it, and let it take care of the multiplexing:

```
open (FH, "| tee file1 file2 file3");
```

Or even:

```
# make STDOUT go to three files, plus original STDOUT
open (STDOUT, "| tee file1 file2 file3") or die "Teeing off: $!\n";
print "whatever\n"                       or die "Writing: $!\n";
close(STDOUT)                            or die "Closing: $!\n";
```

Otherwise you'll have to write your own multiplexing print function -- or your own tee program -- or use Tom Christiansen's, at http://www.perl.com/CPAN/authors/id/TOMC/scripts/tct.gz, which is written in Perl and offers much greater functionality than the stock version.

### How can I read in a file by paragraphs?

Use the `$\` variable (see the *perlvar* manpage for details). You can either set it to `""` to eliminate empty paragraphs (`"abc\n\n\n\ndef"`, for instance, gets treated as two paragraphs and not three), or `"\n\n"` to accept empty paragraphs.

**How can I read a single character from a file? From the keyboard?**

You can use the builtin `getc()` function for most filehandles, but it won't (easily) work on a terminal device. For STDIN, either use the Term::ReadKey module from CPAN, or use the sample code in the `getc` entry in the *perlfunc* manpage.

If your system supports POSIX, you can use the following code, which you'll note turns off echo processing as well.

```
#!/usr/bin/perl -w
use strict;
$| = 1;
for (1..4) {
    my $got;
    print "gimme: ";
    $got = getone();
    print "--> $got\n";
}
exit;

BEGIN {
    use POSIX qw(:termios_h);

    my ($term, $oterm, $echo, $noecho, $fd_stdin);

    $fd_stdin = fileno(STDIN);

    $term     = POSIX::Termios->new();
    $term->getattr($fd_stdin);
    $oterm     = $term->getlflag();

    $echo     = ECHO | ECHOK | ICANON;
    $noecho   = $oterm & ~$echo;

    sub cbreak {
        $term->setlflag($noecho);
        $term->setcc(VTIME, 1);
        $term->setattr($fd_stdin, TCSANOW);
    }

    sub cooked {
        $term->setlflag($oterm);
        $term->setcc(VTIME, 0);
        $term->setattr($fd_stdin, TCSANOW);
    }

    sub getone {
        my $key = '';
        cbreak();
        sysread(STDIN, $key, 1);
        cooked();
        return $key;
    }

}
```

```
END { cooked() }
```

The Term::ReadKey module from CPAN may be easier to use:

```
use Term::ReadKey;
open(TTY, "</dev/tty");
print "Gimme a char: ";
ReadMode "raw";
$key = ReadKey 0, *TTY;
ReadMode "normal";
printf "\nYou said %s, char number %03d\n",
    $key, ord $key;
```

For DOS systems, Dan Carson <dbc@tc.fluke.COM> reports the following:

To put the PC in "raw" mode, use ioctl with some magic numbers gleaned from msdos.c (Perl source file) and Ralf Brown's interrupt list (comes across the net every so often):

```
$old_ioctl = ioctl(STDIN,0,0);     # Gets device info
$old_ioctl &= 0xff;
ioctl(STDIN,1,$old_ioctl | 32);    # Writes it back, setting bit 5
```

Then to read a single character:

```
sysread(STDIN,$c,1);               # Read a single character
```

And to put the PC back to "cooked" mode:

```
ioctl(STDIN,1,$old_ioctl);         # Sets it back to cooked mode.
```

So now you have `$c`. If `ord($c) == 0`, you have a two byte code, which means you hit a special key. Read another byte with `sysread(STDIN,$c,1)`, and that value tells you what combination it was according to this table:

```
# PC 2-byte keycodes = ^@ + the following:

# HEX     KEYS
# ---     ----
# 0F      SHF TAB
# 10-19   ALT QWERTYUIOP
# 1E-26   ALT ASDFGHJKL
# 2C-32   ALT ZXCVBNM
# 3B-44   F1-F10
# 47-49   HOME,UP,PgUp
# 4B      LEFT
# 4D      RIGHT
# 4F-53   END,DOWN,PgDn,Ins,Del
# 54-5D   SHF F1-F10
# 5E-67   CTR F1-F10
# 68-71   ALT F1-F10
# 73-77   CTR LEFT,RIGHT,END,PgDn,HOME
# 78-83   ALT 1234567890-=
# 84      CTR PgUp
```

This is all trial and error I did a long time ago, I hope I'm reading the file that worked.

### How can I tell if there's a character waiting on a filehandle?

The very first thing you should do is look into getting the Term::ReadKey extension from CPAN. It now even has limited support for closed, proprietary (read: not open systems, not POSIX, not Unix, etc) systems.

You should also check out the Frequently Asked Questions list in comp.unix.* for things like this: the answer is essentially the same. It's very system dependent. Here's one solution that works on BSD systems:

```
sub key_ready {
    my($rin, $nfd);
    vec($rin, fileno(STDIN), 1) = 1;
    return $nfd = select($rin,undef,undef,0);
}
```

If you want to find out how many characters are waiting, there's also the FIONREAD ioctl call to be looked at.

The *h2ph* tool that comes with Perl tries to convert C include files to Perl code, which can be `required`. FIONREAD ends up defined as a function in the *sys/ioctl.ph* file:

```
require 'sys/ioctl.ph';

$size = pack("L", 0);
ioctl(FH, FIONREAD(), $size)    or die "Couldn't call ioctl: $!\n";
$size = unpack("L", $size);
```

If *h2ph* wasn't installed or doesn't work for you, you can *grep* the include files by hand:

```
% grep FIONREAD /usr/include/*/*
/usr/include/asm/ioctls.h:#define FIONREAD      0x541B
```

Or write a small C program using the editor of champions:

```
% cat > fionread.c
#include <sys/ioctl.h>
main() {
    printf("%#08x\n", FIONREAD);
}
^D
% cc -o fionread fionread
% ./fionread
0x4004667f
```

And then hard-code it, leaving porting as an exercise to your successor.

```
$FIONREAD = 0x4004667f;         # XXX: opsys dependent

$size = pack("L", 0);
ioctl(FH, $FIONREAD, $size)     or die "Couldn't call ioctl: $!\n";
$size = unpack("L", $size);
```

FIONREAD requires a filehandle connected to a stream, meaning sockets, pipes, and tty devices work, but *not* files.

**How do I do a `tail -f` in perl?**

First try

```
seek(GWFILE, 0, 1);
```

The statement `seek(GWFILE, 0, 1)` doesn't change the current position, but it does clear the end-of-file condition on the handle, so that the next <GWFILE> makes Perl try again to read something.

If that doesn't work (it relies on features of your stdio implementation), then you need something more like this:

```
    for (;;) {
      for ($curpos = tell(GWFILE); <GWFILE>; $curpos = tell(GWFILE)) {
        # search for some stuff and put it into files
      }
      # sleep for a while
      seek(GWFILE, $curpos, 0);  # seek to where we had been
    }
```

If this still doesn't work, look into the POSIX module. POSIX defines the *clearerr()* method, which can remove the end of file condition on a filehandle. The method: read until end of file, *clearerr()*, read some more. Lather, rinse, repeat.

**How do I *dup()* a filehandle in Perl?**

If you check the `open` entry in the *perlfunc* manpage, you'll see that several of the ways to call *open()* should do the trick. For example:

```
open(LOG, ">>/tmp/logfile");
open(STDERR, ">&LOG");
```

Or even with a literal numeric descriptor:

```
$fd = $ENV{MHCONTEXTFD};
open(MHCONTEXT, "<&=$fd");   # like fdopen(3S)
```

Note that "<&STDIN" makes a copy, but "<&=STDIN" make an alias. That means if you close an aliased handle, all aliases become inaccessible. This is not true with a copied one.

Error checking, as always, has been left as an exercise for the reader.

**How do I close a file descriptor by number?**

This should rarely be necessary, as the Perl *close()* function is to be used for things that Perl opened itself, even if it was a dup of a numeric descriptor, as with MHCONTEXT above. But if you really have to, you may be able to do this:

```
require 'sys/syscall.ph';
$rc = syscall(&SYS_close, $fd + 0);  # must force numeric
die "can't sysclose $fd: $!" unless $rc == -1;
```

**Why can't I use "C:\temp\foo" in DOS paths? What doesn't 'C:\temp\foo.exe' work?**

Whoops! You just put a tab and a formfeed into that filename! Remember that within double quoted strings ("like\this"), the backslash is an escape character. The full list of these is in the section on *Quote and Quote-like Operators* in the *perlop* manpage. Unsurprisingly, you don't have a file called "*c:* (tab)*emp* (formfeed)oo" or "*c:* (tab)*emp* (formfeed)oo.exe" on your DOS filesystem.

Either single-quote your strings, or (preferably) use forward slashes. Since all DOS and Windows versions since something like MS−DOS 2.0 or so have treated `/` and `\` the same in a path, you might as well use the one that doesn't clash with Perl -- or the POSIX shell, ANSI C and C++, awk, Tcl, Java, or Python, just to mention a few.

### Why doesn't *glob* ("*.*") get all the files?

Because even on non-Unix ports, Perl's glob function follows standard Unix globbing semantics. You'll need `glob("*")` to get all (non-hidden) files. This makes *glob()* portable.

### Why does Perl let me delete read-only files? Why does `-i` clobber protected files? Isn't this a bug in Perl?

This is elaborately and painstakingly described in the ''Far More Than You Ever Wanted To Know'' in http://www.perl.com/CPAN/doc/FMTEYEWTK/file-dir-perms .

The executive summary: learn how your filesystem works. The permissions on a file say what can happen to the data in that file. The permissions on a directory say what can happen to the list of files in that directory. If you delete a file, you're removing its name from the directory (so the operation depends on the permissions of the directory, not of the file). If you try to write to the file, the permissions of the file govern whether you're allowed to.

### How do I select a random line from a file?

Here's an algorithm from the Camel Book:

```
srand;
rand($.) < 1 && ($line = $_) while <>;
```

This has a significant advantage in space over reading the whole file in. A simple proof by induction is available upon request if you doubt its correctness.

## AUTHOR AND COPYRIGHT

**NAME**

perlfaq6 − Regexps ($Revision: 1.22 $, `$Date`: 1998/07/16 14:01:07 $)

**DESCRIPTION**

This section is surprisingly small because the rest of the FAQ is littered with answers involving regular expressions. For example, decoding a URL and checking whether something is a number are handled with regular expressions, but those answers are found elsewhere in this document (in the section on Data and the Networking one on networking, to be precise).

### How can I hope to use regular expressions without creating illegible and unmaintainable code?

Three techniques can make regular expressions maintainable and understandable.

Comments Outside the Regexp

Describe what you're doing and how you're doing it, using normal Perl comments.

```
# turn the line into the first word, a colon, and the
# number of characters on the rest of the line
s/^(\w+)(.*)/ lc($1) . ":" . length($2) /meg;
```

Comments Inside the Regexp

The `/x` modifier causes whitespace to be ignored in a regexp pattern (except in a character class), and also allows you to use normal comments there, too. As you can imagine, whitespace and comments help a lot.

`/x` lets you turn this:

```
s{<(?:[^>'"]*|".*?"|'.*?')+>}{}gs;
```

into this:

```
s{ <                    # opening angle bracket
    (?:                 # Non-backreffing grouping paren
         [^>'"] *       # 0 or more things that are neither > nor ' nor "
            |           #    or else
         ".*?"          # a section between double quotes (stingy match)
            |           #    or else
         '.*?'          # a section between single quotes (stingy match)
    ) +                 #   all occurring one or more times
   >                    # closing angle bracket
}{}gsx;                 # replace with nothing, i.e. delete
```

It's still not quite so clear as prose, but it is very useful for describing the meaning of each part of the pattern.

Different Delimiters

While we normally think of patterns as being delimited with `/` characters, they can be delimited by almost any character. the *perlre* manpage describes this. For example, the `s///` above uses braces as delimiters. Selecting another delimiter can avoid quoting the delimiter within the pattern:

```
s/\/usr\/local/\/usr\/share/g;      # bad delimiter choice
s#/usr/local#/usr/share#g;          # better
```

### I'm having trouble matching over more than one line. What's wrong?

Either you don't have more than one line in the string you're looking at (probably), or else you aren't using the correct *modifier*(s) on your pattern (possibly).

There are many ways to get multiline data into a string. If you want it to happen automatically while reading input, you'll want to set $/ (probably to '' for paragraphs or `undef` for the whole file) to allow you to read more than one line at a time.

Read the *perlre* manpage to help you decide which of `/s` and `/m` (or both) you might want to use: `/s` allows dot to include newline, and `/m` allows caret and dollar to match next to a newline, not just at the end of the string. You do need to make sure that you've actually got a multiline string in there.

For example, this program detects duplicate words, even when they span line breaks (but not paragraph ones). For this example, we don't need `/s` because we aren't using dot in a regular expression that we want to cross line boundaries. Neither do we need `/m` because we aren't wanting caret or dollar to match at any point inside the record next to newlines. But it's imperative that $/ be set to something other than the default, or else we won't actually ever have a multiline record read in.

```
$/ = '';            # read in more whole paragraph, not just one line
while ( <> ) {
    while ( /\b([\w'-]+)(\s+\1)+\b/gi ) {   # word starts alpha
        print "Duplicate $1 at paragraph $.\n";
    }
}
```

Here's code that finds sentences that begin with "From " (which would be mangled by many mailers):

```
$/ = '';            # read in more whole paragraph, not just one line
while ( <> ) {
    while ( /^From /gm ) { # /m makes ^ match next to \n
        print "leading from in paragraph $.\n";
    }
}
```

Here's code that finds everything between START and END in a paragraph:

```
undef $/;           # read in whole file, not just one line or paragraph
while ( <> ) {
    while ( /START(.*?)END/sm ) { # /s makes . cross line boundaries
        print "$1\n";
    }
}
```

### How can I pull out lines between two patterns that are themselves on different lines?

You can use Perl's somewhat exotic `..` operator (documented in the *perlop* manpage):

```
perl -ne 'print if /START/ .. /END/' file1 file2 ...
```

If you wanted text and not lines, you would use

```
perl -0777 -pe 'print "$1\n" while /START(.*?)END/gs' file1 file2 ...
```

But if you want nested occurrences of `START` through `END`, you'll run up against the problem described in the question in this section on matching balanced text.

Here's another example of using `..`:

```
while (<>) {
    $in_header =   1  .. /^$/;
    $in_body   = /^$/ .. eof();
    # now choose between them
} continue {
    reset if eof();         # fix $.
}
```

**I put a regular expression into $/ but it didn't work. What's wrong?**

$/ must be a string, not a regular expression. Awk has to be better for something. :-)

Actually, you could do this if you don't mind reading the whole file into memory:

```
undef $/;
@records = split /your_pattern/, <FH>;
```

The Net::Telnet module (available from CPAN) has the capability to wait for a pattern in the input stream, or timeout if it doesn't appear within a certain time.

```
## Create a file with three lines.
open FH, ">file";
print FH "The first line\nThe second line\nThe third line\n";
close FH;

## Get a read/write filehandle to it.
$fh = new FileHandle "+<file";

## Attach it to a "stream" object.
use Net::Telnet;
$file = new Net::Telnet (-fhopen => $fh);

## Search for the second line and print out the third.
$file->waitfor('/second line\n/');
print $file->getline;
```

**How do I substitute case insensitively on the LHS, but preserving case on the RHS?**

It depends on what you mean by "preserving case". The following script makes the substitution have the same case, letter by letter, as the original. If the substitution has more characters than the string being substituted, the case of the last character is used for the rest of the substitution.

```
# Original by Nathan Torkington, massaged by Jeffrey Friedl
#
sub preserve_case($$)
{
    my ($old, $new) = @_;
    my ($state) = 0; # 0 = no change; 1 = lc; 2 = uc
    my ($i, $oldlen, $newlen, $c) = (0, length($old), length($new));
    my ($len) = $oldlen < $newlen ? $oldlen : $newlen;
```

```
        for ($i = 0; $i < $len; $i++) {
            if ($c = substr($old, $i, 1), $c =~ /[\W\d_]/) {
                $state = 0;
            } elsif (lc $c eq $c) {
                substr($new, $i, 1) = lc(substr($new, $i, 1));
                $state = 1;
            } else {
                substr($new, $i, 1) = uc(substr($new, $i, 1));
                $state = 2;
            }
        }
        # finish up with any remaining new (for when new is longer than old)
        if ($newlen > $oldlen) {
            if ($state == 1) {
                substr($new, $oldlen) = lc(substr($new, $oldlen));
            } elsif ($state == 2) {
                substr($new, $oldlen) = uc(substr($new, $oldlen));
            }
        }
        return $new;
    }

    $a = "this is a TEsT case";
    $a =~ s/(test)/preserve_case($1, "success")/gie;
    print "$a\n";
```

This prints:

```
    this is a SUcCESS case
```

### How can I make `\w` match national character sets?

See the *perllocale* manpage.

### How can I match a locale-smart version of `/[a-zA-Z]/`?

One alphabetic character would be `/[^\W\d_]/`, no matter what locale you're in. Non-alphabetics would be `/[\W\d_]/` (assuming you don't consider an underscore a letter).

### How can I quote a variable to use in a regexp?

The Perl parser will expand `$variable` and `@variable` references in regular expressions unless the delimiter is a single quote. Remember, too, that the right-hand side of a `s///` substitution is considered a double-quoted string (see the *perlop* manpage for more details). Remember also that any regexp special characters will be acted on unless you precede the substitution with \Q. Here's an example:

```
    $string = "to die?";
    $lhs = "die?";
    $rhs = "sleep no more";

    $string =~ s/\Q$lhs/$rhs/;
    # $string is now "to sleep no more"
```

Without the \Q, the regexp would also spuriously match ‘‘di’’.

### What is `/o` really for?

Using a variable in a regular expression match forces a re-evaluation (and perhaps recompilation) each time through. The `/o` modifier locks in the regexp the first time it's used. This always happens in a constant regular expression, and in fact, the pattern was compiled into the internal format at the same time your entire program was.

Use of `/o` is irrelevant unless variable interpolation is used in the pattern, and if so, the regexp engine will neither know nor care whether the variables change after the pattern is evaluated the *very first* time.

`/o` is often used to gain an extra measure of efficiency by not performing subsequent evaluations when you know it won't matter (because you know the variables won't change), or more rarely, when you don't want the regexp to notice if they do.

For example, here's a "paragrep" program:

```
$/ = '';  # paragraph mode
$pat = shift;
while (<>) {
    print if /$pat/o;
}
```

### How do I use a regular expression to strip C style comments from a file?

While this actually can be done, it's much harder than you'd think. For example, this one-liner

```
perl -0777 -pe 's{/\*.*?\*/}{}gs' foo.c
```

will work in many but not all cases. You see, it's too simple-minded for certain kinds of C programs, in particular, those with what appear to be comments in quoted strings. For that, you'd need something like this, created by Jeffrey Friedl:

```
$/ = undef;
$_ = <>;
s#/\*[^*]*\*+([^/*][^*]*\*+)*/|("(\\.|[^"\\])*"|'(\\.|[^'\\])*'|\n+|.[^/"'\\]*)#$
print;
```

This could, of course, be more legibly written with the `/x` modifier, adding whitespace and comments.

### Can I use Perl regular expressions to match balanced text?

Although Perl regular expressions are more powerful than "mathematical" regular expressions, because they feature conveniences like backreferences (`\1` and its ilk), they still aren't powerful enough. You still need to use non-regexp techniques to parse balanced text, such as the text enclosed between matching parentheses or braces, for example.

An elaborate subroutine (for 7-bit ASCII only) to pull out balanced and possibly nested single chars, like `` ` `` and `'`, `{` and `}`, or `(` and `)` can be found in http://www.perl.com/CPAN/authors/id/TOMC/scripts/pull_quotes.gz .

The C::Scan module from CPAN contains such subs for internal usage, but they are undocumented.

### What does it mean that regexps are greedy? How can I get around it?

Most people mean that greedy regexps match as much as they can. Technically speaking, it's actually the quantifiers (`?`, `*`, `+`, `{}`) that are greedy rather than the whole pattern; Perl prefers local greed and immediate gratification to overall greed. To get non-greedy versions of the same quantifiers, use (`??`, `*?`, `+?`, `{}?`).

An example:

```
    $s1 = $s2 = "I am very very cold";
    $s1 =~ s/ve.*y //;      # I am cold
    $s2 =~ s/ve.*?y //;     # I am very cold
```

Notice how the second substitution stopped matching as soon as it encountered "y ". The `*?` quantifier effectively tells the regular expression engine to find a match as quickly as possible and pass control on to whatever is next in line, like you would if you were playing hot potato.

**How do I process each word on each line?**

Use the split function:

```
while (<>) {
    foreach $word ( split ) {
        # do something with $word here
    }
}
```

Note that this isn't really a word in the English sense; it's just chunks of consecutive non-whitespace characters.

To work with only alphanumeric sequences, you might consider

```
while (<>) {
    foreach $word (m/(\w+)/g) {
        # do something with $word here
    }
}
```

**How can I print out a word-frequency or line-frequency summary?**

To do this, you have to parse out each word in the input stream. We'll pretend that by word you mean chunk of alphabetics, hyphens, or apostrophes, rather than the non-whitespace chunk idea of a word given in the previous question:

```
while (<>) {
    while ( /(\b[^\W_\d][\w'-]+\b)/g ) {   # misses "`sheep'"
        $seen{$1}++;
    }
}
while ( ($word, $count) = each %seen ) {
    print "$count $word\n";
}
```

If you wanted to do the same thing for lines, you wouldn't need a regular expression:

```
while (<>) {
    $seen{$_}++;
}
while ( ($line, $count) = each %seen ) {
    print "$count $line";
}
```

If you want these output in a sorted order, see the section on Hashes.

### How can I do approximate matching?

See the module String::Approx available from CPAN.

### How do I efficiently match many regular expressions at once?

The following is super-inefficient:

```
while (<FH>) {
    foreach $pat (@patterns) {
        if ( /$pat/ ) {
            # do something
        }
    }
}
```

Instead, you either need to use one of the experimental Regexp extension modules from CPAN (which might well be overkill for your purposes), or else put together something like this, inspired from a routine in Jeffrey Friedl's book:

```
sub _bm_build {
    my $condition = shift;
    my @regexp = @_;  # this MUST not be local(); need my()
    my $expr = join $condition => map { "m/\$regexp[$_]/o" } (0..$#regexp);
    my $match_func = eval "sub { $expr }";
    die if $@;  # propagate $@; this shouldn't happen!
    return $match_func;
}

sub bm_and { _bm_build('&&', @_) }
sub bm_or  { _bm_build('||', @_) }

$f1 = bm_and qw{
        xterm
        (?i)window
};

$f2 = bm_or qw{
        \b[Ff]ree\b
        \bBSD\B
        (?i)sys(tem)?\s*[V5]\b
};

# feed me /etc/termcap, prolly
while ( <> ) {
    print "1: $_" if &$f1;
    print "2: $_" if &$f2;
}
```

### Why don't word-boundary searches with `\b` work for me?

Two common misconceptions are that `\b` is a synonym for `\s+`, and that it's the edge between whitespace characters and non-whitespace characters. Neither is correct. `\b` is the place between a `\w` character and a `\W` character (that is, `\b` is the edge of a "word"). It's a zero-width assertion, just like `^`, `$`, and all the other anchors, so it doesn't consume any characters. the *perlre* manpage describes the behaviour of all the regexp metacharacters.

Here are examples of the incorrect application of `\b`, with fixes:

```
"two words" =~ /(\w+)\b(\w+)/;          # WRONG
"two words" =~ /(\w+)\s+(\w+)/;         # right

" =matchless= text" =~ /\b=(\w+)=\b/;   # WRONG
" =matchless= text" =~ /=(\w+)=/;       # right
```

Although they may not do what you thought they did, `\b` and `\B` can still be quite useful. For an example of the correct use of `\b`, see the example of matching duplicate words over multiple lines.

An example of using `\B` is the pattern `\Bis\B`. This will find occurrences of "is" on the insides of words only, as in "thistle", but not "this" or "island".

**Why does using $&, $‘, or $’ slow my program down?**

Because once Perl sees that you need one of these variables anywhere in the program, it has to provide them on each and every pattern match. The same mechanism that handles these provides for the use of `$1`, `$2`, etc., so you pay the same price for each regexp that contains capturing parentheses. But if you never use $&, etc., in your script, then regexps *without* capturing parentheses won't be penalized. So avoid $&, $', and $‘ if you can, but if you can't (and some algorithms really appreciate them), once you've used them once, use them at will, because you've already paid the price.

**What good is** `\G` in a regular expression?

The notation `\G` is used in a match or substitution in conjunction the `/g` modifier (and ignored if there's no `/g`) to anchor the regular expression to the point just past where the last match occurred, i.e. the *pos()* point.

For example, suppose you had a line of text quoted in standard mail and Usenet notation, (that is, with leading > characters), and you want change each leading > into a corresponding `:`. You could do so in this way:

```
 s/^(>+)/':' x length($1)/gem;
```

Or, using `\G`, the much simpler (and faster):

```
s/\G>/:/g;
```

A more sophisticated use might involve a tokenizer. The following lex-like example is courtesy of Jeffrey Friedl. It did not work in 5.003 due to bugs in that release, but does work in 5.004 or better. (Note the use of `/c`, which prevents a failed match with `/g` from resetting the search position back to the beginning of the string.)

```
while (<>) {
  chomp;
  PARSER: {
       m/ \G( \d+\b    )/gcx    && do { print "number: $1\n";  redo; };
       m/ \G( \w+      )/gcx    && do { print "word:   $1\n";  redo; };
       m/ \G( \s+      )/gcx    && do { print "space:  $1\n";  redo; };
       m/ \G( [^\w\d]+ )/gcx    && do { print "other:  $1\n";  redo; };
  }
}
```

Of course, that could have been written as

```
while (<>) {
  chomp;
  PARSER: {
       if ( /\G( \d+\b    )/gcx  {
            print "number: $1\n";
            redo PARSER;
       }
       if ( /\G( \w+      )/gcx  {
            print "word: $1\n";
            redo PARSER;
       }
       if ( /\G( \s+      )/gcx  {
            print "space: $1\n";
            redo PARSER;
       }
       if ( /\G( [^\w\d]+ )/gcx  {
            print "other: $1\n";
            redo PARSER;
       }
  }
}
```

But then you lose the vertical alignment of the regular expressions.

### Are Perl regexps DFAs or NFAs? Are they POSIX compliant?

While it's true that Perl's regular expressions resemble the DFAs (deterministic finite automata) of the *egrep*(1) program, they are in fact implemented as NFAs (non-deterministic finite automata) to allow backtracking and backreferencing. And they aren't POSIX–style either, because those guarantee worst-case behavior for all cases. (It seems that some people prefer guarantees of consistency, even when what's guaranteed is slowness.) See the book "Mastering Regular Expressions" (from O'Reilly) by Jeffrey Friedl for all the details you could ever hope to know on these matters (a full citation appears in the *perlfaq2* manpage).

### What's wrong with using grep or map in a void context?

Both grep and map build a return list, regardless of their context. This means you're making Perl go to the trouble of building up a return list that you then just ignore. That's no way to treat a programming language, you insensitive scoundrel!

### How can I match strings with multibyte characters?

This is hard, and there's no good way. Perl does not directly support wide characters. It pretends that a byte and a character are synonymous. The following set of approaches was offered by Jeffrey Friedl, whose article in issue #5 of The Perl Journal talks about this very matter.

Let's suppose you have some weird Martian encoding where pairs of ASCII uppercase letters encode single Martian letters (i.e. the two bytes "CV" make a single Martian letter, as do the two bytes "SG", "VS", "XX", etc.). Other bytes represent single characters, just like ASCII.

So, the string of Martian "I am CVSGXX!" uses 12 bytes to encode the nine characters 'I', ' ', 'a', 'm', ' ', 'CV', 'SG', 'XX', '!'.

Now, say you want to search for the single character `/GX/`. Perl doesn't know about Martian, so it'll find the two bytes "GX" in the "I am CVSGXX!" string, even though that character isn't there: it just looks like it is because "SG" is next to "XX", but there's no real "GX". This is a big problem.

Here are a few ways, all painful, to deal with it:

```
$martian =~ s/([A-Z][A-Z])/ $1 /g; # Make sure adjacent ``martian'' bytes
                                   # are no longer adjacent.
print "found GX!\n" if $martian =~ /GX/;
```

Or like this:

```
@chars = $martian =~ m/([A-Z][A-Z]|[^A-Z])/g;
# above is conceptually similar to:     @chars = $text =~ m/(.)/g;
#
foreach $char (@chars) {
    print "found GX!\n", last if $char eq 'GX';
}
```

Or like this:

```
while ($martian =~ m/\G([A-Z][A-Z]|.)/gs) {  # \G probably unneeded
    print "found GX!\n", last if $1 eq 'GX';
}
```

Or like this:

```
die "sorry, Perl doesn't (yet) have Martian support )-:\n";
```

In addition, a sample program which converts half-width to full-width katakana (in Shift-JIS or EUC encoding) is available from CPAN as

There are many double- (and multi-) byte encodings commonly used these days. Some versions of these have 1-, 2-, 3-, and 4-byte characters, all mixed.

## AUTHOR AND COPYRIGHT

## NAME

perlfaq7 − Perl Language Issues ($Revision: 1.21 $, `$Date:` 1998/06/22 15:20:07 $)

## DESCRIPTION

This section deals with general Perl language issues that don't clearly fit into any of the other sections.

### Can I get a BNF/yacc/RE for the Perl language?

There is no BNF, but you can paw your way through the yacc grammar in perly.y in the source distribution if you're particularly brave. The grammar relies on very smart tokenizing code, so be prepared to venture into toke.c as well.

In the words of Chaim Frenkel: "Perl's grammar can not be reduced to BNF. The work of parsing perl is distributed between yacc, the lexer, smoke and mirrors."

### What are all these $@%* punctuation signs, and how do I know when to use them?

They are type specifiers, as detailed in the *perldata* manpage:

```
$ for scalar values (number, string or reference)
@ for arrays
% for hashes (associative arrays)
* for all types of that symbol name.  In version 4 you used them like
  pointers, but in modern perls you can just use references.
```

While there are a few places where you don't actually need these type specifiers, you should always use them.

A couple of others that you're likely to encounter that aren't really type specifiers are:

```
<> are used for inputting a record from a filehandle.
\  takes a reference to something.
```

Note that <FILE> is *neither* the type specifier for files nor the name of the handle. It is the <> operator applied to the handle FILE. It reads one line (well, record − see the section on *$/* in the *perlvar* manpage) from the handle FILE in scalar context, or *all* lines in list context. When performing open, close, or any other operation besides <> on files, or even talking about the handle, do *not* use the brackets. These are correct: `eof(FH)`, `seek(FH, 0, 2)` and "copying from STDIN to FILE".

### Do I always/never have to quote my strings or use semicolons and commas?

Normally, a bareword doesn't need to be quoted, but in most cases probably should be (and must be under `use strict`). But a hash key consisting of a simple word (that isn't the name of a defined subroutine) and the left-hand operand to the => operator both count as though they were quoted:

```
This                    is like this
------------            ---------------
$foo{line}              $foo{"line"}
bar => stuff            "bar" => stuff
```

The final semicolon in a block is optional, as is the final comma in a list. Good style (see the *perlstyle* manpage) says to put them in except for one-liners:

```
if ($whoops) { exit 1 }
@nums = (1, 2, 3);
```

```
if ($whoops) {
    exit 1;
}
@lines = (
    "There Beren came from mountains cold",
    "And lost he wandered under leaves",
);
```

**How do I skip some return values?**

One way is to treat the return values as a list and index into it:

```
    $dir = (getpwnam($user))[7];
```

Another way is to use undef as an element on the left-hand-side:

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

**How do I temporarily block warnings?**

The `$^W` variable (documented in the *perlvar* manpage) controls runtime warnings for a block:

```
{
    local $^W = 0;          # temporarily turn off warnings
    $a = $b + $c;           # I know these might be undef
}
```

Note that like all the punctuation variables, you cannot currently use *my()* on `$^W`, only *local()*.

A new `use warnings` pragma is in the works to provide finer control over all this. The curious should check the perl5-porters mailing list archives for details.

**What's an extension?**

A way of calling compiled C code from Perl. Reading the *perlxstut* manpage is a good place to learn more about extensions.

**Why do Perl operators have different precedence than C operators?**

Actually, they don't. All C operators that Perl copies have the same precedence in Perl as they do in C. The problem is with operators that C doesn't have, especially functions that give a list context to everything on their right, eg print, chmod, exec, and so on. Such functions are called "list operators" and appear as such in the precedence table in the *perlop* manpage.

A common mistake is to write:

```
unlink $file || die "snafu";
```

This gets interpreted as:

```
unlink ($file || die "snafu");
```

To avoid this problem, either put in extra parentheses or use the super low precedence `or` operator:

```
(unlink $file) || die "snafu";
unlink $file or die "snafu";
```

The "English" operators (`and`, `or`, `xor`, and `not`) deliberately have precedence lower than that of list operators for just such situations as the one above.

Another operator with surprising precedence is exponentiation. It binds more tightly even than unary minus, making `-2**2` product a negative not a positive four. It is also right-associating, meaning that `2**3**2` is two raised to the ninth power, not eight squared.

Although it has the same precedence as in C, Perl's `?:` operator produces an lvalue. This assigns `$x` to either `$a` or `$b`, depending on the trueness of `$maybe`:

```
($maybe ? $a : $b) = $x;
```

### How do I declare/create a structure?

In general, you don't "declare" a structure. Just use a (probably anonymous) hash reference. See the *perlref* manpage and the *perldsc* manpage for details. Here's an example:

```
$person = {};                   # new anonymous hash
$person->{AGE}  = 24;           # set field AGE to 24
$person->{NAME} = "Nat";        # set field NAME to "Nat"
```

If you're looking for something a bit more rigorous, try the *perltoot* manpage.

### How do I create a module?

A module is a package that lives in a file of the same name. For example, the Hello::There module would live in Hello/There.pm. For details, read the *perlmod* manpage. You'll also find the *Exporter* manpage helpful. If you're writing a C or mixed-language module with both C and Perl, then you should study the *perlxstut* manpage.

Here's a convenient template you might wish you use when starting your own module. Make sure to change the names appropriately.

```
package Some::Module;  # assumes Some/Module.pm

use strict;

BEGIN {
    use Exporter   ();
    use vars       qw($VERSION @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);

    ## set the version for version checking; uncomment to use
    ## $VERSION     = 1.00;

    # if using RCS/CVS, this next line may be preferred,
    # but beware two-digit versions.
    $VERSION = do{my@r=q$Revision: 1.21 $=~/\d+/g;sprintf '%d.'.'%02d'x$#r,@r};

    @ISA         = qw(Exporter);
    @EXPORT      = qw(&func1 &func2 &func3);
    %EXPORT_TAGS = ( );     # eg: TAG => [ qw!name1 name2! ],

    # your exported package globals go here,
    # as well as any optionally exported functions
    @EXPORT_OK   = qw($Var1 %Hashit);
}
use vars      @EXPORT_OK;

# non-exported package globals go here
use vars      qw( @more $stuff );
```

```
# initialize package globals, first exported ones
$Var1   = '';
%Hashit = ();

# then the others (which are still accessible as $Some::Module::stuff)
$stuff  = '';
@more   = ();

# all file-scoped lexicals must be created before
# the functions below that use them.

# file-private lexicals go here
my $priv_var    = '';
my %secret_hash = ();

# here's a file-private function as a closure,
# callable as &$priv_func;  it cannot be prototyped.
my $priv_func = sub {
    # stuff goes here.
};

# make all your functions, whether exported or not;
# remember to put something interesting in the {} stubs
sub func1      {}    # no prototype
sub func2()    {}    # proto'd void
sub func3($$)  {}    # proto'd to 2 scalars

# this one isn't exported, but could be called!
sub func4(\%)  {}    # proto'd to 1 hash ref

END { }       # module clean-up code here (global destructor)

1;            # modules must return true
```

### How do I create a class?

See the *perltoot* manpage for an introduction to classes and objects, as well as the *perlobj* manpage and the *perlbot* manpage.

### How can I tell if a variable is tainted?

See the section on *Laundering and Detecting Tainted Data* in the *perlsec* manpage. Here's an example (which doesn't use any system calls, because the *kill()* is given no processes to signal):

```
sub is_tainted {
    return ! eval { join('',@_), kill 0; 1; };
}
```

This is not `-w` clean, however. There is no `-w` clean way to detect taintedness – take this as a hint that you should untaint all possibly-tainted data.

**What's a closure?**

Closures are documented in the *perlref* manpage.

*Closure* is a computer science term with a precise but hard-to-explain meaning. Closures are implemented in Perl as anonymous subroutines with lasting references to lexical variables outside their own scopes. These lexicals magically refer to the variables that were around when the subroutine was defined (deep binding).

Closures make sense in any programming language where you can have the return value of a function be itself a function, as you can in Perl. Note that some languages provide anonymous functions but are not capable of providing proper closures; the Python language, for example. For more information on closures, check out any textbook on functional programming. Scheme is a language that not only supports but encourages closures.

Here's a classic function-generating function:

```
sub add_function_generator {
  return sub { shift + shift };
}

$add_sub = add_function_generator();
$sum = $add_sub->(4,5);                # $sum is 9 now.
```

The closure works as a *function template* with some customization slots left out to be filled later. The anonymous subroutine returned by *add_function_generator()* isn't technically a closure because it refers to no lexicals outside its own scope.

Contrast this with the following *make_adder()* function, in which the returned anonymous function contains a reference to a lexical variable outside the scope of that function itself. Such a reference requires that Perl return a proper closure, thus locking in for all time the value that the lexical had when the function was created.

```
sub make_adder {
    my $addpiece = shift;
    return sub { shift + $addpiece };
}

$f1 = make_adder(20);
$f2 = make_adder(555);
```

Now `&$f1($n)` is always 20 plus whatever `$n` you pass in, whereas `&$f2($n)` is always 555 plus whatever `$n` you pass in. The `$addpiece` in the closure sticks around.

Closures are often used for less esoteric purposes. For example, when you want to pass in a bit of code into a function:

```
my $line;
timeout( 30, sub { $line = <STDIN> } );
```

If the code to execute had been passed in as a string, `'$line = <STDIN>'`, there would have been no way for the hypothetical *timeout()* function to access the lexical variable `$line` back in its caller's scope.

**What is variable suicide and how can I prevent it?**

Variable suicide is when you (temporarily or permanently) lose the value of a variable. It is caused by scoping through *my()* and *local()* interacting with either closures or aliased *foreach()* interator variables and subroutine arguments. It used to be easy to inadvertently lose a variable's value this way, but now it's much harder. Take this code:

```
my $f = "foo";
sub T {
  while ($i++ < 3) { my $f = $f; $f .= "bar"; print $f, "\n" }
}
T;
print "Finally $f\n";
```

The `$f` that has "bar" added to it three times should be a new `$f` (`my $f` should create a new local variable each time through the loop). It isn't, however. This is a bug, and will be fixed.

**How can I pass/return a {Function, FileHandle, Array, Hash, Method, Regexp}?**

With the exception of regexps, you need to pass references to these objects. See the section on *Pass by Reference* in the *perlsub* manpage for this particular question, and the *perlref* manpage for information on references.

Passing Variables and Functions

Regular variables and functions are quite easy: just pass in a reference to an existing or anonymous variable or function:

```
func( \$some_scalar );

func( \$some_array  );
func( [ 1 .. 10 ]   );

func( \%some_hash   );
func( { this => 10, that => 20 }   );

func( \&some_func   );
func( sub { $_[0] ** $_[1] }   );
```

Passing Filehandles

To pass filehandles to subroutines, use the `*FH` or `\*FH` notations. These are "typeglobs" – see the section on *Typeglobs and Filehandles* in the *perldata* manpage and especially the section on *Pass by Reference* in the *perlsub* manpage for more information.

Here's an excerpt:

If you're passing around filehandles, you could usually just use the bare typeglob, like *STDOUT, but typeglobs references would be better because they'll still work properly under `use strict 'refs'`. For example:

```
splutter(\*STDOUT);
sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}

$rec = get_rec(\*STDIN);
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

If you're planning on generating new filehandles, you could do this:

```
sub openit {
    my $name = shift;
    local *FH;
    return open (FH, $path) ? *FH : undef;
}
$fh = openit('< /etc/motd');
print <$fh>;
```

### Passing Regexps

To pass regexps around, you'll need to either use one of the highly experimental regular expression modules from CPAN (Nick Ing-Simmons's Regexp or Ilya Zakharevich's Devel::Regexp), pass around strings and use an exception-trapping eval, or else be be very, very clever. Here's an example of how to pass in a string to be regexp compared:

```
sub compare($$) {
    my ($val1, $regexp) = @_;
    my $retval = eval { $val =~ /$regexp/ };
    die if $@;
    return $retval;
}

$match = compare("old McDonald", q/d.*D/);
```

Make sure you never say something like this:

```
return eval "\$val =~ /$regexp/";   # WRONG
```

or someone can sneak shell escapes into the regexp due to the double interpolation of the eval and the double-quoted string. For example:

```
$pattern_of_evil = 'danger ${ system("rm -rf * &") } danger';

eval "\$string =~ /$pattern_of_evil/";
```

Those preferring to be very, very clever might see the O'Reilly book, *Mastering Regular Expressions*, by Jeffrey Friedl. Page 273's *Build_MatchMany_Function()* is particularly interesting. A complete citation of this book is given in the *perlfaq2* manpage.

### Passing Methods

To pass an object method into a subroutine, you can do this:

```
call_a_lot(10, $some_obj, "methname")
sub call_a_lot {
    my ($count, $widget, $trick) = @_;
    for (my $i = 0; $i < $count; $i++) {
        $widget->$trick();
    }
}
```

Or you can use a closure to bundle up the object and its method call and arguments:

```
my $whatnot =  sub { $some_obj->obfuscate(@args) };
func($whatnot);
sub func {
    my $code = shift;
    &$code();
}
```

You could also investigate the *can()* method in the UNIVERSAL class (part of the standard perl distribution).

### How do I create a static variable?

As with most things in Perl, TMTOWTDI. What is a "static variable" in other languages could be either a function-private variable (visible only within a single function, retaining its value between calls to that function), or a file-private variable (visible only to functions within the file it was declared in) in Perl.

Here's code to implement a function-private variable:

```
BEGIN {
    my $counter = 42;
    sub prev_counter { return --$counter }
    sub next_counter { return $counter++ }
}
```

Now *prev_counter()* and *next_counter()* share a private variable `$counter` that was initialized at compile time.

To declare a file-private variable, you'll still use a *my()*, putting it at the outer scope level at the top of the file. Assume this is in file Pax.pm:

```
package Pax;
my $started = scalar(localtime(time()));

sub begun { return $started }
```

When `use Pax` or `require Pax` loads this module, the variable will be initialized. It won't get garbage-collected the way most variables going out of scope do, because the *begun()* function cares about it, but no one else can get it. It is not called `$Pax::started` because its scope is unrelated to the package. It's scoped to the file. You could conceivably have several packages in that same file all accessing the same private variable, but another file with the same package couldn't get to it.

See the section on *Peristent Private Variables* in the *perlsub* manpage for details.

### What's the difference between dynamic and lexical (static) scoping? Between *local()* and *my()*?

`local($x)` saves away the old value of the global variable `$x`, and assigns a new value for the duration of the subroutine, *which is visible in other functions called from that subroutine*. This is done at run-time, so is called dynamic scoping. *local()* always affects global variables, also called package variables or dynamic variables.

`my($x)` creates a new variable that is only visible in the current subroutine. This is done at compile-time, so is called lexical or static scoping. *my()* always affects private variables, also called lexical variables or (improperly) *static*(ly scoped) variables.

For instance:

```
sub visible {
    print "var has value $var\n";
}
```

```
sub dynamic {
    local $var = 'local';    # new temporary value for the still-global
    visible();               #   variable called $var
}

sub lexical {
    my $var = 'private';     # new private variable, $var
    visible();               # (invisible outside of sub scope)
}

$var = 'global';

visible();                   # prints global
dynamic();                   # prints local
lexical();                   # prints global
```

Notice how at no point does the value "private" get printed. That's because `$var` only has that value within the block of the *lexical()* function, and it is hidden from called subroutine.

In summary, *local()* doesn't make what you think of as private, local variables. It gives a global variable a temporary value. *my()* is what you're looking for if you want private variables.

See the section on *Private Variables via my()* in the *perlsub* manpage and the section on *Temporary Values via local()* in the *perlsub* manpage for excruciating details.

### How can I access a dynamic variable while a similarly named lexical is in scope?

You can do this via symbolic references, provided you haven't set `use strict "refs"`. So instead of `$var`, use `${'var'}`.

```
local $var = "global";
my    $var = "lexical";

print "lexical is $var\n";

no strict 'refs';
print "global  is ${'var'}\n";
```

If you know your package, you can just mention it explicitly, as in `$Some_Pack::var`. Note that the notation `$::var` is *not* the dynamic `$var` in the current package, but rather the one in the `main` package, as though you had written `$main::var`. Specifying the package directly makes you hard-code its name, but it executes faster and avoids running afoul of `use strict "refs"`.

### What's the difference between deep and shallow binding?

In deep binding, lexical variables mentioned in anonymous subroutines are the same ones that were in scope when the subroutine was created. In shallow binding, they are whichever variables with the same names happen to be in scope when the subroutine is called. Perl always uses deep binding of lexical variables (i.e., those created with *my()*). However, dynamic variables (aka global, local, or package variables) are effectively shallowly bound. Consider this just one more reason not to use them. See the answer to the section on *What's a closure?*.

### Why doesn't "*my* ($foo) = <FILE>;" work right?

`my()` and `local()` give list context to the right hand side of =. The <FH> read operation, like so many of Perl's functions and operators, can tell which context it was called in and behaves appropriately. In general, the *scalar()* function can help. This function does nothing to the data itself (contrary to popular myth) but rather tells its argument to behave in whatever its scalar fashion is. If that function doesn't have a defined scalar behavior, this of course doesn't help you (such as with *sort()*).

To enforce scalar context in this particular case, however, you need merely omit the parentheses:

```
local($foo) = <FILE>;           # WRONG
local($foo) = scalar(<FILE>);   # ok
local $foo  = <FILE>;           # right
```

You should probably be using lexical variables anyway, although the issue is the same here:

```
my($foo) = <FILE>;  # WRONG
my $foo  = <FILE>;  # right
```

### How do I redefine a builtin function, operator, or method?

Why do you want to do that? :–)

If you want to override a predefined function, such as *open()*, then you'll have to import the new definition from a different module. See the section on *Overriding Builtin Functions* in the *perlsub* manpage. There's also an example in the section on *Class::Template* in the *perltoot* manpage.

If you want to overload a Perl operator, such as + or `**`, then you'll want to use the `use overload` pragma, documented in the *overload* manpage.

If you're talking about obscuring method calls in parent classes, see the section on *Overridden Methods* in the *perltoot* manpage.

### What's the difference between calling a function as &foo and *foo()*?

When you call a function as `&foo`, you allow that function access to your current `@_` values, and you bypass prototypes. That means that the function doesn't get an empty `@_`, it gets yours! While not strictly speaking a bug (it's documented that way in the *perlsub* manpage), it would be hard to consider this a feature in most cases.

When you call your function as `&foo()`, then you *do* get a new `@_`, but prototyping is still circumvented.

Normally, you want to call a function using `foo()`. You may only omit the parentheses if the function is already known to the compiler because it already saw the definition (`use` but not `require`), or via a forward reference or `use subs` declaration. Even in this case, you get a clean `@_` without any of the old values leaking through where they don't belong.

### How do I create a switch or case statement?

This is explained in more depth in the the *perlsyn* manpage. Briefly, there's no official case statement, because of the variety of tests possible in Perl (numeric comparison, string comparison, glob comparison, regexp matching, overloaded comparisons, ...). Larry couldn't decide how best to do this, so he left it out, even though it's been on the wish list since perl1.

The general answer is to write a construct like this:

```
for ($variable_to_test) {
    if    (/pat1/)  { }     # do something
    elsif (/pat2/)  { }     # do something else
    elsif (/pat3/)  { }     # do something else
    else            { }     # default
}
```

Here's a simple example of a switch based on pattern matching, this time lined up in a way to make it look more like a switch statement. We'll do a multi-way conditional based on the type of reference stored in `$whatchamacallit`:

```
SWITCH: for (ref $whatchamacallit) {

    /^$/            && die "not a reference";

    /SCALAR/        && do {
                            print_scalar($$ref);
                            last SWITCH;
                    };

    /ARRAY/         && do {
                            print_array(@$ref);
                            last SWITCH;
                    };

    /HASH/          && do {
                            print_hash(%$ref);
                            last SWITCH;
                    };

    /CODE/          && do {
                            warn "can't print function ref";
                            last SWITCH;
                    };

    # DEFAULT

    warn "User defined type skipped";

}
```

See `perlsyn/"Basic BLOCKs and Switch Statements"` for many other examples in this style.

Sometimes you should change the positions of the constant and the variable. For example, let's say you wanted to test which of many answers you were given, but in a case-insensitive way that also allows abbreviations. You can use the following technique if the strings all start with different characters, or if you want to arrange the matches so that one takes precedence over another, as `"SEND"` has precedence over `"STOP"` here:

```
chomp($answer = <>);
if    ("SEND"  =~ /^\Q$answer/i) { print "Action is send\n"  }
elsif ("STOP"  =~ /^\Q$answer/i) { print "Action is stop\n"  }
elsif ("ABORT" =~ /^\Q$answer/i) { print "Action is abort\n" }
elsif ("LIST"  =~ /^\Q$answer/i) { print "Action is list\n"  }
elsif ("EDIT"  =~ /^\Q$answer/i) { print "Action is edit\n"  }
```

A totally different approach is to create a hash of function references.

```
my %commands = (
    "happy" => \&joy,
    "sad",  => \&sullen,
    "done"  => sub { die "See ya!" },
    "mad"   => \&angry,
);
```

```
print "How are you? ";
chomp($string = <STDIN>);
if ($commands{$string}) {
    $commands{$string}->();
} else {
    print "No such command: $string\n";
}
```

### How can I catch accesses to undefined variables/functions/methods?

The AUTOLOAD method, discussed in the section on *Autoloading* in the *perlsub* manpage and the section on *AUTOLOAD: Proxy Methods* in the *perltoot* manpage, lets you capture calls to undefined functions and methods.

When it comes to undefined variables that would trigger a warning under `-w`, you can use a handler to trap the pseudo-signal `__WARN__` like this:

```
$SIG{__WARN__} = sub {

    for ( $_[0] ) {          # voici un switch statement

        /Use of uninitialized value/  && do {
            # promote warning to a fatal
            die $_;
        };

        # other warning cases to catch could go here;

        warn $_;
    }

};
```

### Why can't a method included in this same file be found?

Some possible reasons: your inheritance is getting confused, you've misspelled the method name, or the object is of the wrong type. Check out the *perltoot* manpage for details on these. You may also use `print ref($object)` to find out the class `$object` was blessed into.

Another possible reason for problems is because you've used the indirect object syntax (eg, `find Guru "Samy"`) on a class name before Perl has seen that such a package exists. It's wisest to make sure your packages are all defined before you start using them, which will be taken care of if you use the `use` statement instead of `require`. If not, make sure to use arrow notation (eg, `Guru->find("Samy")`) instead. Object notation is explained in the *perlobj* manpage.

Make sure to read about creating modules in the *perlmod* manpage and the perils of indirect objects in the section on *WARNING* in the *perlobj* manpage.

### How can I find out my current package?

If you're just a random program, you can do this to find out what the currently compiled package is:

```
my $packname = __PACKAGE__;
```

But if you're a method and you want to print an error message that includes the kind of object you were called on (which is not necessarily the same as the one in which you were compiled):

```
sub amethod {
    my $self  = shift;
    my $class = ref($self) || $self;
    warn "called me from a $class object";
}
```

### How can I comment out a large block of perl code?

Use embedded POD to discard it:

```
# program is here

=for nobody
This paragraph is commented out

# program continues

=begin comment text

all of this stuff

here will be ignored
by everyone

=end comment text

=cut
```

This can't go just anywhere. You have to put a pod directive where the parser is expecting a new statement, not just in the middle of an expression or some other arbitrary yacc grammar production.

## AUTHOR AND COPYRIGHT



Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

**NAME**

perlfaq8 − System Interaction ($Revision: 1.25 $, `$Date:` 1998/07/05 15:07:20 $)

**DESCRIPTION**

This section of the Perl FAQ covers questions involving operating system interaction. This involves interprocess communication (IPC), control over the user-interface (keyboard, screen and pointing devices), and most anything else not related to data manipulation.

Read the FAQs and documentation specific to the port of perl to your operating system (eg, the *perlvms* manpage, the *perlplan9* manpage, ...). These should contain more detailed information on the vagaries of your perl.

### How do I find out which operating system I'm running under?

The $^O variable ($OSNAME if you use English) contains the operating system that your perl binary was built for.

### How come *exec()* doesn't return?

Because that's what it does: it replaces your currently running program with a different one. If you want to keep going (as is probably the case if you're asking this question) use *system()* instead.

### How do I do fancy stuff with the keyboard/screen/mouse?

How you access/control keyboards, screens, and pointing devices ("mice") is system-dependent. Try the following modules:

Keyboard

```
Term::Cap                   Standard perl distribution
Term::ReadKey               CPAN
Term::ReadLine::Gnu         CPAN
Term::ReadLine::Perl        CPAN
Term::Screen                CPAN
```

Screen

```
Term::Cap                   Standard perl distribution
Curses                      CPAN
Term::ANSIColor             CPAN
```

Mouse

```
Tk                          CPAN
```

Some of these specific cases are shown below.

### How do I print something out in color?

In general, you don't, because you don't know whether the recipient has a color-aware display device. If you know that they have an ANSI terminal that understands color, you can use the Term::ANSIColor module from CPAN:

```
use Term::ANSIColor;
print color("red"), "Stop!\n", color("reset");
print color("green"), "Go!\n", color("reset");
```

Or like this:

```
use Term::ANSIColor qw(:constants);
print RED, "Stop!\n", RESET;
print GREEN, "Go!\n", RESET;
```

**How do I read just one key without waiting for a return key?**

Controlling input buffering is a remarkably system-dependent matter. If most systems, you can just use the **stty** command as shown in the `getc` entry in the *perlfunc* manpage, but as you see, that's already getting you into portability snags.

```
open(TTY, "+</dev/tty") or die "no tty: $!";
system "stty  cbreak </dev/tty >/dev/tty 2>&1";
$key = getc(TTY);            # perhaps this works
# OR ELSE
sysread(TTY, $key, 1);       # probably this does
system "stty -cbreak </dev/tty >/dev/tty 2>&1";
```

The Term::ReadKey module from CPAN offers an easy-to-use interface that should be more efficient than shelling out to **stty** for each key. It even includes limited support for Windows.

```
use Term::ReadKey;
ReadMode('cbreak');
$key = ReadKey(0);
ReadMode('normal');
```

However, that requires that you have a working C compiler and can use it to build and install a CPAN module. Here's a solution using the standard POSIX module, which is already on your systems (assuming your system supports POSIX).

```
use HotKey;
$key = readkey();
```

And here's the HotKey module, which hides the somewhat mystifying calls to manipulate the POSIX termios structures.

```
# HotKey.pm
package HotKey;

@ISA = qw(Exporter);
@EXPORT = qw(cbreak cooked readkey);

use strict;
use POSIX qw(:termios_h);
my ($term, $oterm, $echo, $noecho, $fd_stdin);

$fd_stdin = fileno(STDIN);
$term     = POSIX::Termios->new();
$term->getattr($fd_stdin);
$oterm     = $term->getlflag();

$echo     = ECHO | ECHOK | ICANON;
$noecho   = $oterm & ~$echo;
```

```
sub cbreak {
    $term->setlflag($noecho);  # ok, so i don't want echo either
    $term->setcc(VTIME, 1);
    $term->setattr($fd_stdin, TCSANOW);
}

sub cooked {
    $term->setlflag($oterm);
    $term->setcc(VTIME, 0);
    $term->setattr($fd_stdin, TCSANOW);
}

sub readkey {
    my $key = '';
    cbreak();
    sysread(STDIN, $key, 1);
    cooked();
    return $key;
}

END { cooked() }

1;
```

**How do I check whether input is ready on the keyboard?**

The easiest way to do this is to read a key in nonblocking mode with the Term::ReadKey module from CPAN, passing it an argument of −1 to indicate not to block:

```
use Term::ReadKey;

ReadMode('cbreak');

if (defined ($char = ReadKey(-1)) ) {
    # input was waiting and it was $char
} else {
    # no input was waiting
}

ReadMode('normal');                  # restore normal tty settings
```

**How do I clear the screen?**

If you only have to so infrequently, use `system`:

```
system("clear");
```

If you have to do this a lot, save the clear string so you can print it 100 times without calling a program 100 times:

```
$clear_string = `clear`;
print $clear_string;
```

If you're planning on doing other screen manipulations, like cursor positions, etc, you might wish to use Term::Cap module:

```
use Term::Cap;
$terminal = Term::Cap->Tgetent( {OSPEED => 9600} );
$clear_string = $terminal->Tputs('cl');
```

### How do I get the screen size?

If you have Term::ReadKey module installed from CPAN, you can use it to fetch the width and height in characters and in pixels:

```
use Term::ReadKey;
($wchar, $hchar, $wpixels, $hpixels) = GetTerminalSize();
```

This is more portable than the raw `ioctl`, but not as illustrative:

```
require 'sys/ioctl.ph';
die "no TIOCGWINSZ " unless defined &TIOCGWINSZ;
open(TTY, "+</dev/tty")                     or die "No tty: $!";
unless (ioctl(TTY, &TIOCGWINSZ, $winsize='')) {
    die sprintf "$0: ioctl TIOCGWINSZ (%08x: $!)\n", &TIOCGWINSZ;
}
($row, $col, $xpixel, $ypixel) = unpack('S4', $winsize);
print "(row,col) = ($row,$col)";
print "  (xpixel,ypixel) = ($xpixel,$ypixel)" if $xpixel || $ypixel;
print "\n";
```

### How do I ask the user for a password?

(This question has nothing to do with the web. See a different FAQ for that.)

There's an example of this in the `crypt` entry in the *perlfunc* manpage). First, you put the terminal into "no echo" mode, then just read the password normally. You may do this with an old-style *ioctl()* function, POSIX terminal control (see the *POSIX* manpage, and Chapter 7 of the Camel), or a call to the **stty** program, with varying degrees of portability.

You can also do this for most systems using the Term::ReadKey module from CPAN, which is easier to use and in theory more portable.

```
use Term::ReadKey;

ReadMode('noecho');
$password = ReadLine(0);
```

### How do I read and write the serial port?

This depends on which operating system your program is running on. In the case of Unix, the serial ports will be accessible through files in /dev; on other systems, the devices names will doubtless differ. Several problem areas common to all device interaction are the following

lockfiles
: Your system may use lockfiles to control multiple access. Make sure you follow the correct protocol. Unpredictable behaviour can result from multiple processes reading from one device.

open mode
: If you expect to use both read and write operations on the device, you'll have to open it for update (see the section on *open* in the *perlfunc* manpage for details). You may wish to open it without running the risk of blocking by using *sysopen()* and `O_RDWR|O_NDELAY|O_NOCTTY` from the Fcntl module (part of the standard perl distribution). See the section on *sysopen* in the *perlfunc* manpage for more on this approach.

end of line

Some devices will be expecting a “\r” at the end of each line rather than a “\n”. In some ports of perl, “\r” and “\n” are different from their usual (Unix) ASCII values of “\012” and “\015”. You may have to give the numeric values you want directly, using octal (”\015"), hex ("0x0D"), or as a control-character specification (”\cM").

```
print DEV "atv1\012";        # wrong, for some devices
print DEV "atv1\015";        # right, for some devices
```

Even though with normal text files, a “\n” will do the trick, there is still no unified scheme for terminating a line that is portable between Unix, DOS/Win, and Macintosh, except to terminate *ALL* line ends with “\015\012”, and strip what you don’t need from the output. This applies especially to socket I/O and autoflushing, discussed next.

flushing output

If you expect characters to get to your device when you *print()* them, you’ll want to autoflush that filehandle. You can use *select()* and the `$|` variable to control autoflushing (see the section on *$|* in the *perlvar* manpage and the `select` entry in the *perlfunc* manpage):

```
$oldh = select(DEV);
$| = 1;
select($oldh);
```

You’ll also see code that does this without a temporary variable, as in

```
select((select(DEV), $| = 1)[0]);
```

Or if you don’t mind pulling in a few thousand lines of code just because you’re afraid of a little `$|` variable:

```
use IO::Handle;
DEV->autoflush(1);
```

As mentioned in the previous item, this still doesn’t work when using socket I/O between Unix and Macintosh. You’ll need to hardcode your line terminators, in that case.

non-blocking input

If you are doing a blocking *read()* or *sysread()*, you’ll have to arrange for an alarm handler to provide a timeout (see the `alarm` entry in the *perlfunc* manpage). If you have a non-blocking open, you’ll likely have a non-blocking read, which means you may have to use a 4-arg *select()* to determine whether I/O is ready on that device (see the section on *select* in the *perlfunc* manpage.

While trying to read from his caller-id box, the notorious Jamie Zawinski <jwz@netscape.com>, after much gnashing of teeth and fighting with sysread, sysopen, POSIX’s tcgetattr business, and various other functions that go bump in the night, finally came up with this:

```
sub open_modem {
    use IPC::Open2;
    my $stty = `/bin/stty -g`;
    open2( \*MODEM_IN, \*MODEM_OUT, "cu -l$modem_device -s2400 2>&1");
    # starting cu hoses /dev/tty's stty settings, even when it has
    # been opened on a pipe...
    system("/bin/stty $stty");
    $_ = <MODEM_IN>;
    chop;
    if ( !m/^Connected/ ) {
        print STDERR "$0: cu printed `$_' instead of `Connected'\n";
    }
}
```

### How do I decode encrypted password files?

You spend lots and lots of money on dedicated hardware, but this is bound to get you talked about.

Seriously, you can't if they are Unix password files – the Unix password system employs one-way encryption. It's more like hashing than encryption. The best you can check is whether something else hashes to the same string. You can't turn a hash back into the original string. Programs like Crack can forcibly (and intelligently) try to guess passwords, but don't (can't) guarantee quick success.

If you're worried about users selecting bad passwords, you should proactively check when they try to change their password (by modifying *passwd*(1), for example).

### How do I start a process in the background?

You could use

```
system("cmd &")
```

or you could use fork as documented in the section on *fork* in the *perlfunc* manpage, with further examples in the *perlipc* manpage. Some things to be aware of, if you're on a Unix-like system:

STDIN, STDOUT, and STDERR are shared
: Both the main process and the backgrounded one (the "child" process) share the same STDIN, STDOUT and STDERR filehandles. If both try to access them at once, strange things can happen. You may want to close or reopen these for the child. You can get around this with `open`ing a pipe (see the section on *open* in the *perlfunc* manpage) but on some systems this means that the child process cannot outlive the parent.

Signals
: You'll have to catch the SIGCHLD signal, and possibly SIGPIPE too. SIGCHLD is sent when the backgrounded process finishes. SIGPIPE is sent when you write to a filehandle whose child process has closed (an untrapped SIGPIPE can cause your program to silently die). This is not an issue with `system("cmd&")`.

Zombies
: You have to be prepared to "reap" the child process when it finishes

```
$SIG{CHLD} = sub { wait };
```

See the section on *Signals* in the *perlipc* manpage for other examples of code to do this. Zombies are not an issue with `system("prog &")`.

### How do I trap control characters/signals?

You don't actually ''trap'' a control character. Instead, that character generates a signal which is sent to your terminal's currently foregrounded process group, which you then trap in your process. Signals are documented in the section on *Signals* in the *perlipc* manpage and chapter 6 of the Camel.

Be warned that very few C libraries are re-entrant. Therefore, if you attempt to *print()* in a handler that got invoked during another stdio operation your internal structures will likely be in an inconsistent state, and your program will dump core. You can sometimes avoid this by using *syswrite()* instead of *print()*.

Unless you're exceedingly careful, the only safe things to do inside a signal handler are: set a variable and exit. And in the first case, you should only set a variable in such a way that *malloc()* is not called (eg, by setting a variable that already has a value).

For example:

```
$Interrupted = 0;   # to ensure it has a value
$SIG{INT} = sub {
    $Interrupted++;
    syswrite(STDERR, "ouch\n", 5);
}
```

However, because syscalls restart by default, you'll find that if you're in a ''slow'' call, such as <FH>, *read()*, *connect()*, or *wait()*, that the only way to terminate them is by ''longjumping'' out; that is, by raising an exception. See the time-out handler for a blocking *flock()* in the section on *Signals* in the *perlipc* manpage or chapter 6 of the Camel.

### How do I modify the shadow password file on a Unix system?

If perl was installed correctly, and your shadow library was written properly, the getpw*() functions described in the *perlfunc* manpage should in theory provide (read-only) access to entries in the shadow password file. To change the file, make a new shadow password file (the format varies from system to system – see the *passwd(5)* manpage for specifics) and use *pwd_mkdb* (8) to install it (see the *pwd_mkdb(5)* manpage for more details).

### How do I set the time and date?

Assuming you're running under sufficient permissions, you should be able to set the system-wide date and time by running the *date* (1) program. (There is no way to set the time and date on a per-process basis.) This mechanism will work for Unix, MS–DOS, Windows, and NT; the VMS equivalent is `set time`.

However, if all you want to do is change your timezone, you can probably get away with setting an environment variable:

```
$ENV{TZ} = "MST7MDT";                  # unixish
$ENV{'SYS$TIMEZONE_DIFFERENTIAL'}="-5" # vms
system "trn comp.lang.perl.misc";
```

### How can I *sleep()* or *alarm()* for under a second?

If you want finer granularity than the 1 second that the *sleep()* function provides, the easiest way is to use the *select()* function as documented in the section on *select* in the *perlfunc* manpage. If your system has itimers and *syscall()* support, you can check out the old example in http://www.perl.com/CPAN/doc/misc/ancient/tutorial/eg/itimers.pl .

**How can I measure time under a second?**

In general, you may not be able to. The Time::HiRes module (available from CPAN) provides this functionality for some systems.

In general, you may not be able to. But if your system supports both the *syscall()* function in Perl as well as a system call like *gettimeofday* (2), then you may be able to do something like this:

```
require 'sys/syscall.ph';

$TIMEVAL_T = "LL";

$done = $start = pack($TIMEVAL_T, ());

syscall( &SYS_gettimeofday, $start, 0)) != -1
           or die "gettimeofday: $!";

   ##########################
   # DO YOUR OPERATION HERE #
   ##########################

syscall( &SYS_gettimeofday, $done, 0) != -1
       or die "gettimeofday: $!";

@start = unpack($TIMEVAL_T, $start);
@done  = unpack($TIMEVAL_T, $done);

# fix microseconds
for ($done[1], $start[1]) { $_ /= 1_000_000 }

$delta_time = sprintf "%.4f", ($done[0]  + $done[1]  )
                                            -
                             ($start[0] + $start[1] );
```

**How can I do an** *atexit()* or *setjmp()*/*longjmp()*? (Exception handling)

Release 5 of Perl added the END block, which can be used to simulate *atexit()*. Each package's END block is called when the program or thread ends (see the *perlmod* manpage manpage for more details).

For example, you can use this to make sure your filter program managed to finish its output without filling up the disk:

```
END {
    close(STDOUT) || die "stdout close failed: $!";
}
```

The END block isn't called when untrapped signals kill the program, though, so if you use END blocks you should also use

```
    use sigtrap qw(die normal-signals);
```

Perl's exception-handling mechanism is its *eval()* operator. You can use *eval()* as setjmp and *die()* as longjmp. For details of this, see the section on signals, especially the time-out handler for a blocking *flock()* in the section on *Signals* in the *perlipc* manpage and chapter 6 of the Camel.

If exception handling is all you're interested in, try the exceptions.pl library (part of the standard perl distribution).

If you want the *atexit()* syntax (and an *rmexit()* as well), try the AtExit module available from CPAN.

### Why doesn't my sockets program work under System V (Solaris)? What does the error message "Protocol not supported" mean?

Some Sys-V based systems, notably Solaris 2.X, redefined some of the standard socket constants. Since these were constant across all architectures, they were often hardwired into perl code. The proper way to deal with this is to "use Socket" to get the correct values.

Note that even though SunOS and Solaris are binary compatible, these values are different. Go figure.

### How can I call my system's unique C functions from Perl?

In most cases, you write an external module to do it – see the answer to "Where can I learn about linking C with Perl? [h2xs, xsubpp]". However, if the function is a system call, and your system supports *syscall()*, you can use the syscall function (documented in the *perlfunc* manpage).

Remember to check the modules that came with your distribution, and CPAN as well – someone may already have written a module to do it.

### Where do I get the include files to do *ioctl()* or *syscall()*?

Historically, these would be generated by the h2ph tool, part of the standard perl distribution. This program converts *cpp* (1) directives in C header files to files containing subroutine definitions, like &SYS_getitimer, which you can use as arguments to your functions. It doesn't work perfectly, but it usually gets most of the job done. Simple files like *errno.h*, *syscall.h*, and *socket.h* were fine, but the hard ones like *ioctl.h* nearly always need to hand-edited. Here's how to install the *.ph files:

```
1.  become super-user
2.  cd /usr/include
3.  h2ph *.h */*.h
```

If your system supports dynamic loading, for reasons of portability and sanity you probably ought to use h2xs (also part of the standard perl distribution). This tool converts C header files to Perl extensions. See the *perlxstut* manpage for how to get started with h2xs.

If your system doesn't support dynamic loading, you still probably ought to use h2xs. See the *perlxstut* manpage and the *ExtUtils::MakeMaker* manpage for more information (in brief, just use **make perl** instead of a plain **make** to rebuild perl with a new static extension).

### Why do setuid perl scripts complain about kernel problems?

Some operating systems have bugs in the kernel that make setuid scripts inherently insecure. Perl gives you a number of options (described in the *perlsec* manpage) to work around such systems.

### How can I open a pipe both to and from a command?

The IPC::Open2 module (part of the standard perl distribution) is an easy-to-use approach that internally uses *pipe()*, *fork()*, and *exec()* to do the job. Make sure you read the deadlock warnings in its documentation, though (see the *IPC::Open2* manpage). See the section on *Bidirectional Communication with Another Process* in the *perlipc* manpage and the section on *Bidirectional Communication with Yourself* in the *perlipc* manpage

You may also use the IPC::Open3 module (part of the standard perl distribution), but be warned that it has a different order of arguments from IPC::Open2 (see the *IPC::Open3* manpage).

### Why can't I get the output of a command with *system()*?

You're confusing the purpose of *system()* and backticks (``). *system()* runs a command and returns exit status information (as a 16 bit value: the low 7 bits are the signal the process died from, if any, and the high 8 bits are the actual exit value). Backticks (``) run a command and return what it sent to STDOUT.

```
$exit_status   = system("mail-users");
$output_string = `ls`;
```

**How can I capture STDERR from an external command?**

There are three basic ways of running external commands:

```
system $cmd;                # using system()
$output = `$cmd`;           # using backticks (``)
open (PIPE, "cmd |");       # using open()
```

With *system()*, both STDOUT and STDERR will go the same place as the script's versions of these, unless the command redirects them. Backticks and *open()* read **only** the STDOUT of your command.

With any of these, you can change file descriptors before the call:

```
open(STDOUT, ">logfile");
system("ls");
```

or you can use Bourne shell file-descriptor redirection:

```
$output = `$cmd 2>some_file`;
open (PIPE, "cmd 2>some_file |");
```

You can also use file-descriptor redirection to make STDERR a duplicate of STDOUT:

```
$output = `$cmd 2>&1`;
open (PIPE, "cmd 2>&1 |");
```

Note that you *cannot* simply open STDERR to be a dup of STDOUT in your Perl program and avoid calling the shell to do the redirection. This doesn't work:

```
open(STDERR, ">&STDOUT");
$alloutput = `cmd args`;  # stderr still escapes
```

This fails because the *open()* makes STDERR go to where STDOUT was going at the time of the *open()*. The backticks then make STDOUT go to a string, but don't change STDERR (which still goes to the old STDOUT).

Note that you *must* use Bourne shell (*sh*(1)) redirection syntax in backticks, not *csh*(1)! Details on why Perl's *system()* and backtick and pipe opens all use the Bourne shell are in http://www.perl.com/CPAN/doc/FMTEYEWTK/versus/csh.whynot . To capture a command's STDERR and STDOUT together:

```
$output = `cmd 2>&1`;                          # either with backticks
$pid = open(PH, "cmd 2>&1 |");                 # or with an open pipe
while (<PH>) { }                               #    plus a read
```

To capture a command's STDOUT but discard its STDERR:

```
$output = `cmd 2>/dev/null`;                   # either with backticks
$pid = open(PH, "cmd 2>/dev/null |");          # or with an open pipe
while (<PH>) { }                               #    plus a read
```

To capture a command's STDERR but discard its STDOUT:

```
$output = `cmd 2>&1 1>/dev/null`;              # either with backticks
$pid = open(PH, "cmd 2>&1 1>/dev/null |");     # or with an open pipe
while (<PH>) { }                               #    plus a read
```

To exchange a command's STDOUT and STDERR in order to capture the STDERR but leave its STDOUT to come out our old STDERR:

```
$output = `cmd 3>&1 1>&2 2>&3 3>&-`;        # either with backticks
$pid = open(PH, "cmd 3>&1 1>&2 2>&3 3>&-|");# or with an open pipe
while (<PH>) { }                            #    plus a read
```

To read both a command's STDOUT and its STDERR separately, it's easiest and safest to redirect them separately to files, and then read from those files when the program is done:

```
system("program args 1>/tmp/program.stdout 2>/tmp/program.stderr");
```

Ordering is important in all these examples. That's because the shell processes file descriptor redirections in strictly left to right order.

```
system("prog args 1>tmpfile 2>&1");
system("prog args 2>&1 1>tmpfile");
```

The first command sends both standard out and standard error to the temporary file. The second command sends only the old standard output there, and the old standard error shows up on the old standard out.

**Why doesn't** *open()* return an error when a pipe open fails?

It does, but probably not how you expect it to. On systems that follow the standard *fork()*/*exec()* paradigm (such as Unix), it works like this: *open()* causes a *fork()*. In the parent, *open()* returns with the process ID of the child. The child *exec()*s the command to be piped to/from. The parent can't know whether the *exec()* was successful or not – all it can return is whether the *fork()* succeeded or not. To find out if the command succeeded, you have to catch SIGCHLD and *wait()* to get the exit status. You should also catch SIGPIPE if you're writing to the child -- you may not have found out the *exec()* failed by the time you write. This is documented in the *perlipc* manpage.

On systems that follow the *spawn()* paradigm, *open() might* do what you expect – unless perl uses a shell to start your command. In this case the *fork()*/*exec()* description still applies.

**What's wrong with using backticks in a void context?**

Strictly speaking, nothing. Stylistically speaking, it's not a good way to write maintainable code because backticks have a (potentially humungous) return value, and you're ignoring it. It's may also not be very efficient, because you have to read in all the lines of output, allocate memory for them, and then throw it away. Too often people are lulled to writing:

```
`cp file file.bak`;
```

And now they think "Hey, I'll just always use backticks to run programs." Bad idea: backticks are for capturing a program's output; the *system()* function is for running programs.

Consider this line:

```
`cat /etc/termcap`;
```

You haven't assigned the output anywhere, so it just wastes memory (for a little while). Plus you forgot to check `$?` to see whether the program even ran correctly. Even if you wrote

```
print `cat /etc/termcap`;
```

In most cases, this could and probably should be written as

```
system("cat /etc/termcap") == 0
    or die "cat program failed!";
```

Which will get the output quickly (as its generated, instead of only at the end) and also check the return value.

*system()* also provides direct control over whether shell wildcard processing may take place, whereas backticks do not.

### How can I call backticks without shell processing?

This is a bit tricky. Instead of writing

```
@ok = `grep @opts '$search_string' @filenames`;
```

You have to do this:

```
my @ok = ();
if (open(GREP, "-|")) {
    while (<GREP>) {
        chomp;
        push(@ok, $_);
    }
    close GREP;
} else {
    exec 'grep', @opts, $search_string, @filenames;
}
```

Just as with *system()*, no shell escapes happen when you *exec()* a list.

There are more examples of this the section on *Safe Pipe Opens* in the *perlipc* manpage.

### Why can't my script read from STDIN after I gave it EOF (^D on Unix, ^Z on MS−DOS)?

Because some stdio's set error and eof flags that need clearing. The POSIX module defines *clearerr()* that you can use. That is the technically correct way to do it. Here are some less reliable workarounds:

1. Try keeping around the seekpointer and go there, like this:

   ```
   $where = tell(LOG);
   seek(LOG, $where, 0);
   ```

2. If that doesn't work, try seeking to a different part of the file and then back.
3. If that doesn't work, try seeking to a different part of the file, reading something, and then seeking back.
4. If that doesn't work, give up on your stdio package and use sysread.

### How can I convert my shell script to perl?

Learn Perl and rewrite it. Seriously, there's no simple converter. Things that are awkward to do in the shell are easy to do in Perl, and this very awkwardness is what would make a shell->perl converter nigh-on impossible to write. By rewriting it, you'll think about what you're really trying to do, and hopefully will escape the shell's pipeline datastream paradigm, which while convenient for some matters, causes many inefficiencies.

### Can I use perl to run a telnet or ftp session?

Try the Net::FTP, TCP::Client, and Net::Telnet modules (available from CPAN). http://www.perl.com/CPAN/scripts/netstuff/telnet.emul.shar will also help for emulating the telnet protocol, but Net::Telnet is quite probably easier to use..

If all you want to do is pretend to be telnet but don't need the initial telnet handshaking, then the standard dual-process approach will suffice:

```
use IO::Socket;             # new in 5.004
$handle = IO::Socket::INET->new('www.perl.com:80')
        || die "can't connect to port 80 on www.perl.com: $!";
$handle->autoflush(1);
if (fork()) {               # XXX: undef means failure
    select($handle);
    print while <STDIN>;    # everything from stdin to socket
} else {
    print while <$handle>;  # everything from socket to stdout
}
close $handle;
exit;
```

### How can I write expect in Perl?

Once upon a time, there was a library called chat2.pl (part of the standard perl distribution), which never really got finished. If you find it somewhere, *don't use it*. These days, your best bet is to look at the Expect module available from CPAN, which also requires two other modules from CPAN, IO::Pty and IO::Stty.

### Is there a way to hide perl's command line from programs such as "ps"?

First of all note that if you're doing this for security reasons (to avoid people seeing passwords, for example) then you should rewrite your program so that critical information is never given as an argument. Hiding the arguments won't make your program completely secure.

To actually alter the visible command line, you can assign to the variable `$0` as documented in the *perlvar* manpage. This won't work on all operating systems, though. Daemon programs like sendmail place their state there, as in:

```
$0 = "orcus [accepting connections]";
```

### I {changed directory, modified my environment} in a perl script. How come the change disappeared when I exited the script? How do I get my changes to be visible?

Unix

In the strictest sense, it can't be done -- the script executes as a different process from the shell it was started from. Changes to a process are not reflected in its parent, only in its own children created after the change. There is shell magic that may allow you to fake it by *eval()*ing the script's output in your shell; check out the comp.unix.questions FAQ for details.

### How do I close a process's filehandle without waiting for it to complete?

Assuming your system supports such things, just send an appropriate signal to the process (see the section on *kill* in the *perlfunc* manpage. It's common to first send a TERM signal, wait a little bit, and then send a KILL signal to finish it off.

### How do I fork a daemon process?

If by daemon process you mean one that's detached (disassociated from its tty), then the following process is reported to work on most Unixish systems. Non-Unix users should check their Your_OS::Process module for other solutions.

- Open /dev/tty and use the the TIOCNOTTY ioctl on it. See the *tty(4)* manpage for details. Or better yet, you can just use the *POSIX::setsid()* function, so you don't have to worry about process groups.

- Change directory to /
- Reopen STDIN, STDOUT, and STDERR so they're not connected to the old tty.
- Background yourself like this:

  ```
  fork && exit;
  ```

**How do I make my program run with sh and csh?**

See the *eg/nih* script (part of the perl source distribution).

**How do I find out if I'm running interactively or not?**

Good question. Sometimes `-t STDIN` and `-t STDOUT` can give clues, sometimes not.

```
if (-t STDIN && -t STDOUT) {
    print "Now what? ";
}
```

On POSIX systems, you can test whether your own process group matches the current process group of your controlling terminal as follows:

```
use POSIX qw/getpgrp tcgetpgrp/;
open(TTY, "/dev/tty") or die $!;
$tpgrp = tcgetpgrp(TTY);
$pgrp = getpgrp();
if ($tpgrp == $pgrp) {
    print "foreground\n";
} else {
    print "background\n";
}
```

**How do I timeout a slow event?**

Use the *alarm()* function, probably in conjunction with a signal handler, as documented the section on *Signals* in the *perlipc* manpage and chapter 6 of the Camel. You may instead use the more flexible Sys::AlarmCall module available from CPAN.

**How do I set CPU limits?**

Use the BSD::Resource module from CPAN.

**How do I avoid zombies on a Unix system?**

Use the reaper code from the section on *Signals* in the *perlipc* manpage to call *wait()* when a SIGCHLD is received, or else use the double-fork technique described in the `fork` entry in the *perlfunc* manpage.

**How do I use an SQL database?**

There are a number of excellent interfaces to SQL databases. See the DBD::* modules available from http://www.perl.com/CPAN/modules/dbperl/DBD . A lot of information on this can be found at http://www.hermetica.com/technologia/perl/DBI/index.html .

**How do I make a** *system()* exit on control-C?

You can't. You need to imitate the *system()* call (see the *perlipc* manpage for sample code) and then have a signal handler for the INT signal that passes the signal on to the subprocess. Or you can check for it:

```
$rc = system($cmd);
if ($rc & 127) { die "signal death" }
```

**How do I open a file without blocking?**

If you're lucky enough to be using a system that supports non-blocking reads (most Unixish systems do), you need only to use the O_NDELAY or O_NONBLOCK flag from the Fcntl module in conjunction with *sysopen()*:

```
use Fcntl;
sysopen(FH, "/tmp/somefile", O_WRONLY|O_NDELAY|O_CREAT, 0644)
    or die "can't open /tmp/somefile: $!":
```

**How do I install a CPAN module?**

The easiest way is to have the CPAN module do it for you. This module comes with perl version 5.004 and later. To manually install the CPAN module, or any well-behaved CPAN module for that matter, follow these steps:

1 Unpack the source into a temporary area.

2

```
perl Makefile.PL
```

3

```
make
```

4

```
make test
```

5

```
make install
```

If your version of perl is compiled without dynamic loading, then you just need to replace step 3 (**make**) with **make perl** and you will get a new *perl* binary with your extension linked in.

See the *ExtUtils::MakeMaker* manpage for more details on building extensions. See also the next question.

**What's the difference between require and use?**

Perl offers several different ways to include code from one file into another. Here are the deltas between the various inclusion constructs:

```
1)  do $file is like eval `cat $file`, except the former:
    1.1: searches @INC.
    1.2: bequeaths an *unrelated* lexical scope on the eval'ed code.

2)  require $file is like do $file, except the former:
    2.1: checks for redundant loading, skipping already loaded files.
    2.2: raises an exception on failure to find, compile, or execute $file.
```

```
3)  require Module is like require "Module.pm", except the former:
    3.1: translates each "::" into your system's directory separator.
    3.2: primes the parser to disambiguate class Module as an indirect object.

4)  use Module is like require Module, except the former:
    4.1: loads the module at compile time, not run-time.
    4.2: imports symbols and semantics from that package to the current one.
```

In general, you usually want `use` and a proper Perl module.

### How do I keep my own module/library directory?

When you build modules, use the PREFIX option when generating Makefiles:

```
perl Makefile.PL PREFIX=/u/mydir/perl
```

then either set the PERL5LIB environment variable before you run scripts that use the modules/libraries (see the *perlrun* manpage) or say

```
use lib '/u/mydir/perl';
```

See Perl's the *lib* manpage for more information.

### How do I add the directory my program lives in to the module/library search path?

```
use FindBin;
use lib "$FindBin::Bin";
use your_own_modules;
```

### How do I add a directory to my include path at runtime?

Here are the suggested ways of modifying your include path:

```
the PERLLIB environment variable
the PERL5LIB environment variable
the perl -Idir commpand line flag
the use lib pragma, as in
    use lib "$ENV{HOME}/myown_perllib";
```

The latter is particularly useful because it knows about machine dependent architectures. The lib.pm pragmatic module was first included with the 5.002 release of Perl.

## AUTHOR AND COPYRIGHT

**NAME**

perlfaq9 – Networking ($Revision: 1.20 $, `$Date`: 1998/06/22 18:31:09 $)

**DESCRIPTION**

This section deals with questions related to networking, the internet, and a few on the web.

**My CGI script runs from the command line but not the browser. (500 Server Error)**

If you can demonstrate that you've read the following FAQs and that your problem isn't something simple that can be easily answered, you'll probably receive a courteous and useful reply to your question if you post it on comp.infosystems.www.authoring.cgi (if it's something to do with HTTP, HTML, or the CGI protocols). Questions that appear to be Perl questions but are really CGI ones that are posted to comp.lang.perl.misc may not be so well received.

The useful FAQs and related documents are:

```
CGI FAQ
    http://www.webthing.com/page.cgi/cgifaq

Web FAQ
    http://www.boutell.com/faq/

WWW Security FAQ
    http://www.w3.org/Security/Faq/

HTTP Spec
    http://www.w3.org/pub/WWW/Protocols/HTTP/

HTML Spec
    http://www.w3.org/TR/REC-html40/
    http://www.w3.org/pub/WWW/MarkUp/

CGI Spec
    http://www.w3.org/CGI/

CGI Security FAQ
    http://www.go2net.com/people/paulp/cgi-security/safe-cgi.txt
```

**How can I get better error messages from a CGI program?**

Use the CGI::Carp module. It replaces `warn` and `die`, plus the normal Carp modules `carp`, `croak`, and `confess` functions with more verbose and safer versions. It still sends them to the normal server error log.

```
use CGI::Carp;
warn "This is a complaint";
die "But this one is serious";
```

The following use of CGI::Carp also redirects errors to a file of your choice, placed in a BEGIN block to catch compile-time warnings as well:

```
BEGIN {
    use CGI::Carp qw(carpout);
    open(LOG, ">>/var/local/cgi-logs/mycgi-log")
        or die "Unable to append to mycgi-log: $!\n";
    carpout(*LOG);
}
```

You can even arrange for fatal errors to go back to the client browser, which is nice for your own debugging, but might confuse the end user.

```
use CGI::Carp qw(fatalsToBrowser);
die "Bad error here";
```

Even if the error happens before you get the HTTP header out, the module will try to take care of this to avoid the dreaded server 500 errors. Normal warnings still go out to the server error log (or wherever you've sent them with `carpout`) with the application name and date stamp prepended.

**How do I remove HTML from a string?**

The most correct way (albeit not the fastest) is to use HTML::Parse from CPAN (part of the libwww-perl distribution, which is a must-have module for all web hackers).

Many folks attempt a simple-minded regular expression approach, like `s/<.*?>//g`, but that fails in many cases because the tags may continue over line breaks, they may contain quoted angle-brackets, or HTML comment may be present. Plus folks forget to convert entities, like `<` for example.

Here's one "simple-minded" approach, that works for most files:

```
#!/usr/bin/perl -p0777
s/<(?:[^>'"]*|(['"]).*?\1)*>//gs
```

If you want a more complete solution, see the 3-stage striphtml program in http://www.perl.com/CPAN/authors/Tom_Christiansen/scripts/striphtml.gz .

Here are some tricky cases that you should think about when picking a solution:

```
<IMG SRC = "foo.gif" ALT = "A > B">

<IMG SRC = "foo.gif"
     ALT = "A > B">

<!-- <A comment> -->

<script>if (a<b && a>c)</script>

<# Just data #>

<![INCLUDE CDATA [ >>>>>>>>>>>> ]]>
```

If HTML comments include other tags, those solutions would also break on text like this:

```
<!-- This section commented out.
    <B>You can't see me!</B>
-->
```

**How do I extract URLs?**

A quick but imperfect approach is

```
#!/usr/bin/perl -n00
# qxurl - tchrist@perl.com
print "$2\n" while m{
    < \s*
      A \s+ HREF \s* = \s* (["']) (.*?) \1
    \s* >
}gsix;
```

This version does not adjust relative URLs, understand alternate bases, deal with HTML comments, deal with HREF and NAME attributes in the same tag, or accept URLs themselves as arguments. It also runs about 100x faster than a more "complete" solution using the LWP suite of modules, such as the http://www.perl.com/CPAN/authors/Tom_Christiansen/scripts/xurl.gz program.

### How do I download a file from the user's machine? How do I open a file on another machine?

In the context of an HTML form, you can use what's known as **multipart/form-data** encoding. The CGI.pm module (available from CPAN) supports this in the *start_multipart_form()* method, which isn't the same as the *startform()* method.

### How do I make a pop-up menu in HTML?

Use the **<SELECT>** and **<OPTION>** tags. The CGI.pm module (available from CPAN) supports this widget, as well as many others, including some that it cleverly synthesizes on its own.

### How do I fetch an HTML file?

One approach, if you have the lynx text-based HTML browser installed on your system, is this:

```
$html_code = `lynx -source $url`;
$text_data = `lynx -dump $url`;
```

The libwww-perl (LWP) modules from CPAN provide a more powerful way to do this. They work through proxies, and don't require lynx:

```
# simplest version
use LWP::Simple;
$content = get($URL);

# or print HTML from a URL
use LWP::Simple;
getprint "http://www.sn.no/libwww-perl/";

# or print ASCII from HTML from a URL
use LWP::Simple;
use HTML::Parse;
use HTML::FormatText;
my ($html, $ascii);
$html = get("http://www.perl.com/");
defined $html
    or die "Can't fetch HTML from http://www.perl.com/";
$ascii = HTML::FormatText->new->format(parse_html($html));
print $ascii;
```

### How do I automate an HTML form submission?

If you're submitting values using the GET method, create a URL and encode the form using the `query_form` method:

```
use LWP::Simple;
use URI::URL;

my $url = url('http://www.perl.com/cgi-bin/cpan_mod');
$url->query_form(module => 'DB_File', readme => 1);
$content = get($url);
```

If you're using the POST method, create your own user agent and encode the content appropriately.

```
use HTTP::Request::Common qw(POST);
use LWP::UserAgent;

$ua = LWP::UserAgent->new();
my $req = POST 'http://www.perl.com/cgi-bin/cpan_mod',
               [ module => 'DB_File', readme => 1 ];
$content = $ua->request($req)->as_string;
```

### How do I decode or create those %−encodings on the web?

Here's an example of decoding:

```
$string = "http://altavista.digital.com/cgi-bin/query?pg=q&what=news&fmt=.&q=%2B
$string =~ s/%([a-fA-F0-9]{2})/chr(hex($1))/ge;
```

Encoding is a bit harder, because you can't just blindly change all the non-alphanumunder character (`\W`) into their hex escapes. It's important that characters with special meaning like `/` and `?` *not* be translated. Probably the easiest way to get this right is to avoid reinventing the wheel and just use the URI::Escape module, which is part of the libwww-perl package (LWP) available from CPAN.

### How do I redirect to another page?

Instead of sending back a `Content-Type` as the headers of your reply, send back a `Location:` header. Officially this should be a `URI:` header, so the CGI.pm module (available from CPAN) sends back both:

```
Location: http://www.domain.com/newpage
URI: http://www.domain.com/newpage
```

Note that relative URLs in these headers can cause strange effects because of "optimizations" that servers do.

```
$url = "http://www.perl.com/CPAN/";
print "Location: $url\n\n";
exit;
```

To be correct to the spec, each of those `"\n"` should really each be `"\015\012"`, but unless you're stuck on MacOS, you probably won't notice.

### How do I put a password on my web pages?

That depends. You'll need to read the documentation for your web server, or perhaps check some of the other FAQs referenced above.

### How do I edit my .htpasswd and .htgroup files with Perl?

The HTTPD::UserAdmin and HTTPD::GroupAdmin modules provide a consistent OO interface to these files, regardless of how they're stored. Databases may be text, dbm, Berkley DB or any database with a DBI compatible driver. HTTPD::UserAdmin supports files used by the 'Basic' and 'Digest' authentication schemes. Here's an example:

```
use HTTPD::UserAdmin ();
HTTPD::UserAdmin
      ->new(DB => "/foo/.htpasswd")
      ->add($username => $password);
```

### How do I make sure users can't enter values into a form that cause my CGI script to do bad things?

Read the CGI security FAQ, at http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html, and the Perl/CGI FAQ at http://www.perl.com/CPAN/doc/FAQs/cgi/perl-cgi-faq.html.

In brief: use tainting (see the *perlsec* manpage), which makes sure that data from outside your script (eg, CGI parameters) are never used in `eval` or `system` calls. In addition to tainting, never use the single-argument form of *system()* or *exec()*. Instead, supply the command and arguments as a list, which prevents shell globbing.

### How do I parse a mail header?

For a quick-and-dirty solution, try this solution derived from page 222 of the 2nd edition of ''Programming Perl":

```
$/ = '';
$header = <MSG>;
$header =~ s/\n\s+/ /g;      # merge continuation lines
%head = ( UNIX_FROM_LINE, split /^([-\w]+):\s*/m, $header );
```

That solution doesn't do well if, for example, you're trying to maintain all the Received lines. A more complete approach is to use the Mail::Header module from CPAN (part of the MailTools package).

### How do I decode a CGI form?

You use a standard module, probably CGI.pm. Under no circumstances should you attempt to do so by hand!

You'll see a lot of CGI programs that blindly read from STDIN the number of bytes equal to CONTENT_LENGTH for POSTs, or grab QUERY_STRING for decoding GETs. These programs are very poorly written. They only work sometimes. They typically forget to check the return value of the *read()* system call, which is a cardinal sin. They don't handle HEAD requests. They don't handle multipart forms used for file uploads. They don't deal with GET/POST combinations where query fields are in more than one place. They don't deal with keywords in the query string.

In short, they're bad hacks. Resist them at all costs. Please do not be tempted to reinvent the wheel. Instead, use the CGI.pm or CGI_Lite.pm (available from CPAN), or if you're trapped in the module-free land of perl1 .. perl4, you might look into cgi-lib.pl (available from http://www.bio.cam.ac.uk/web/form.html).

Make sure you know whether to use a GET or a POST in your form. GETs should only be used for something that doesn't update the server. Otherwise you can get mangled databases and repeated feedback mail messages. The fancy word for this is ''idempotency''. This simply means that there should be no difference between making a GET request for a particular URL once or multiple times. This is because the HTTP protocol definition says that a GET request may be cached by the browser, or server, or an intervening proxy. POST requests cannot be cached, because each request is independent and matters. Typically, POST requests change or depend on state on the server (query or update a database, send mail, or purchase a computer).

### How do I check a valid mail address?

You can't, at least, not in real time. Bummer, eh?

Without sending mail to the address and seeing whether there's a human on the other hand to answer you, you cannot determine whether a mail address is valid. Even if you apply the mail header standard, you can have problems, because there are deliverable addresses that aren't RFC−822 (the mail header standard) compliant, and addresses that aren't deliverable which are compliant.

Many are tempted to try to eliminate many frequently-invalid mail addresses with a simple regexp, such as `/^[\w.-]+\@([\w.-]\.)+\w+$/`. It's a very bad idea. However, this also throws out many valid ones, and says nothing about potential deliverability, so is not suggested. Instead, see http://www.perl.com/CPAN/authors/Tom_Christiansen/scripts/ckaddr.gz , which actually checks against the full RFC spec (except for nested comments), looks for addresses you may not wish to accept mail to (say, Bill Clinton or your postmaster), and then makes sure that the hostname given can be looked up in the DNS MX records. It's not fast, but it works for what it tries to do.

Our best advice for verifying a person's mail address is to have them enter their address twice, just as you normally do to change a password. This usually weeds out typos. If both versions match, send mail to that address with a personal message that looks somewhat like:

```
Dear someuser@host.com,

Please confirm the mail address you gave us Wed May  6 09:38:41
MDT 1998 by replying to this message.  Include the string
"Rumpelstiltskin" in that reply, but spelled in reverse; that is,
start with "Nik...".  Once this is done, your confirmed address will
be entered into our records.
```

If you get the message back and they've followed your directions, you can be reasonably assured that it's real.

A related strategy that's less open to forgery is to give them a PIN (personal ID number). Record the address and PIN (best that it be a random one) for later processing. In the mail you send, ask them to include the PIN in their reply. But if it bounces, or the message is included via a ‘‘vacation’’ script, it'll be there anyway. So it's best to ask them to mail back a slight alteration of the PIN, such as with the characters reversed, one added or subtracted to each digit, etc.

### How do I decode a MIME/BASE64 string?

The MIME−tools package (available from CPAN) handles this and a lot more. Decoding BASE64 becomes as simple as:

```
use MIME::base64;
$decoded = decode_base64($encoded);
```

A more direct approach is to use the *unpack()* function's ‘‘u’’ format after minor transliterations:

```
tr#A-Za-z0-9+/##cd;                   # remove non-base64 chars
tr#A-Za-z0-9+/# -_#;                  # convert to uuencoded format
$len = pack("c", 32 + 0.75*length);   # compute length byte
print unpack("u", $len . $_);         # uudecode and print
```

### How do I return the user's mail address?

On systems that support getpwuid, the $< variable and the Sys::Hostname module (which is part of the standard perl distribution), you can probably try using something like this:

```
use Sys::Hostname;
$address = sprintf('%s@%s', getpwuid($<), hostname);
```

Company policies on mail address can mean that this generates addresses that the company's mail system will not accept, so you should ask for users' mail addresses when this matters. Furthermore, not all systems on which Perl runs are so forthcoming with this information as is Unix.

The Mail::Util module from CPAN (part of the MailTools package) provides a *mailaddress()* function that tries to guess the mail address of the user. It makes a more intelligent guess than the code above, using information given when the module was installed, but it could still be incorrect. Again, the best way is often just to ask the user.

### How do I send mail?

Use the `sendmail` program directly:

```
open(SENDMAIL, "|/usr/lib/sendmail -oi -t -odq")
                    or die "Can't fork for sendmail: $!\n";
print SENDMAIL <<"EOF";
From: User Originating Mail <me\@host>
To: Final Destination <you\@otherhost>
Subject: A relevant subject line

Body of the message goes here, in as many lines as you like.
EOF
close(SENDMAIL)     or warn "sendmail didn't close nicely";
```

The **–oi** option prevents sendmail from interpreting a line consisting of a single dot as "end of message". The **–t** option says to use the headers to decide who to send the message to, and **–odq** says to put the message into the queue. This last option means your message won't be immediately delivered, so leave it out if you want immediate delivery.

Or use the CPAN module Mail::Mailer:

```
use Mail::Mailer;

$mailer = Mail::Mailer->new();
$mailer->open({ From    => $from_address,
                To      => $to_address,
                Subject => $subject,
              })
    or die "Can't open: $!\n";
print $mailer $body;
$mailer->close();
```

The Mail::Internet module uses Net::SMTP which is less Unix-centric than Mail::Mailer, but less reliable. Avoid raw SMTP commands. There are many reasons to use a mail transport agent like sendmail. These include queueing, MX records, and security.

### How do I read mail?

Use the Mail::Folder module from CPAN (part of the MailFolder package) or the Mail::Internet module from CPAN (also part of the MailTools package).

```
# sending mail
use Mail::Internet;
use Mail::Header;
# say which mail host to use
$ENV{SMTPHOSTS} = 'mail.frii.com';
# create headers
$header = new Mail::Header;
$header->add('From', 'gnat@frii.com');
$header->add('Subject', 'Testing');
$header->add('To', 'gnat@frii.com');
# create body
$body = 'This is a test, ignore';
# create mail object
$mail = new Mail::Internet(undef, Header => $header, Body => \[$body]);
# send it
$mail->smtpsend or die;
```

Often a module is overkill, though. Here's a mail sorter.

```
#!/usr/bin/perl
# bysub1 - simple sort by subject
my(@msgs, @sub);
my $msgno = -1;
$/ = '';                        # paragraph reads
while (<>) {
    if (/^From/m) {
        /^Subject:\s*(?:Re:\s*)*(.*)/mi;
        $sub[++$msgno] = lc($1) || '';
    }
    $msgs[$msgno] .= $_;
}
for my $i (sort { $sub[$a] cmp $sub[$b] || $a <=> $b } (0 .. $#msgs)) {
    print $msgs[$i];
}
```

Or more succinctly,

```
#!/usr/bin/perl -n00
# bysub2 - awkish sort-by-subject
BEGIN { $msgno = -1 }
$sub[++$msgno] = (/^Subject:\s*(?:Re:\s*)*(.*)/mi)[0] if /^From/m;
$msg[$msgno] .= $_;
END { print @msg[ sort { $sub[$a] cmp $sub[$b] || $a <=> $b } (0 .. $#msg) ] }
```

### How do I find out my hostname/domainname/IP address?

The normal way to find your own hostname is to call the `` `hostname` `` program. While sometimes expedient, this has some problems, such as not knowing whether you've got the canonical name or not. It's one of those tradeoffs of convenience versus portability.

The Sys::Hostname module (part of the standard perl distribution) will give you the hostname after which you can find out the IP address (assuming you have working DNS) with a *gethostbyname()* call.

```
use Socket;
use Sys::Hostname;
my $host = hostname();
my $addr = inet_ntoa(scalar(gethostbyname($name)) || 'localhost');
```

Probably the simplest way to learn your DNS domain name is to grok it out of /etc/resolv.conf, at least under Unix. Of course, this assumes several things about your resolv.conf configuration, including that it exists.

(We still need a good DNS domain name-learning method for non-Unix systems.)

### How do I fetch a news article or the active newsgroups?

Use the Net::NNTP or News::NNTPClient modules, both available from CPAN. This can make tasks like fetching the newsgroup list as simple as:

```
perl -MNews::NNTPClient
  -e 'print News::NNTPClient->new->list("newsgroups")'
```

### How do I fetch/put an FTP file?

LWP::Simple (available from CPAN) can fetch but not put. Net::FTP (also available from CPAN) is more complex but can put as well as fetch.

### How can I do RPC in Perl?

A DCE::RPC module is being developed (but is not yet available), and will be released as part of the DCE−Perl package (available from CPAN). No ONC::RPC module is known.

## AUTHOR AND COPYRIGHT



Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

**NAME**

a2p − Awk to Perl translator

**SYNOPSIS**

**a2p [options] filename**

**DESCRIPTION**

*A2p* takes an awk script specified on the command line (or from standard input) and produces a comparable *perl* script on the standard output.

### Options

Options include:

**−D<number>**
sets debugging flags.

**−F<character>**
tells a2p that this awk script is always invoked with this **−F** switch.

**−n<fieldlist>**
specifies the names of the input fields if input does not have to be split into an array. If you were translating an awk script that processes the password file, you might say:

```
a2p -7 -nlogin.password.uid.gid.gcos.shell.home
```

Any delimiter can be used to separate the field names.

**−<number>**
causes a2p to assume that input will always have that many fields.

**−o** tells a2p to use old awk behavior. For now, the only difference is that old awk always has a line loop, even if there are no line actions, whereas new awk does not.

### "Considerations"

A2p cannot do as good a job translating as a human would, but it usually does pretty well. There are some areas where you may want to examine the perl script produced and tweak it some. Here are some of them, in no particular order.

There is an awk idiom of putting *int()* around a string expression to force numeric interpretation, even though the argument is always integer anyway. This is generally unneeded in perl, but a2p can't tell if the argument is always going to be integer, so it leaves it in. You may wish to remove it.

Perl differentiates numeric comparison from string comparison. Awk has one operator for both that decides at run time which comparison to do. A2p does not try to do a complete job of awk emulation at this point. Instead it guesses which one you want. It's almost always right, but it can be spoofed. All such guesses are marked with the comment "`#???`". You should go through and check them. You might want to run at least once with the **−w** switch to perl, which will warn you if you use == where you should have used eq.

Perl does not attempt to emulate the behavior of awk in which nonexistent array elements spring into existence simply by being referenced. If somehow you are relying on this mechanism to create null entries for a subsequent for...in, they won't be there in perl.

If a2p makes a split line that assigns to a list of variables that looks like (Fld1, Fld2, Fld3...) you may want to rerun a2p using the **−n** option mentioned above. This will let you name the fields throughout the script. If it splits to an array instead, the script is probably referring to the number of fields somewhere.

The exit statement in awk doesn't necessarily exit; it goes to the END block if there is one. Awk scripts that do contortions within the END block to bypass the block under such circumstances can be simplified by removing the conditional in the END block and just exiting directly from the perl script.

Perl has two kinds of array, numerically-indexed and associative. Perl associative arrays are called "hashes". Awk arrays are usually translated to hashes, but if you happen to know that the index is always going to be numeric you could change the {...} to [...]. Iteration over a hash is done using the *keys()* function, but iteration over an array is NOT. You might need to modify any loop that iterates over such an array.

Awk starts by assuming OFMT has the value %.6g. Perl starts by assuming its equivalent, $#, to have the value %.20g. You'll want to set $# explicitly if you use the default value of OFMT.

Near the top of the line loop will be the split operation that is implicit in the awk script. There are times when you can move this down past some conditionals that test the entire record so that the split is not done as often.

For aesthetic reasons you may wish to change the array base $[ from 1 back to perl's default of 0, but remember to change all array subscripts AND all *substr()* and *index()* operations to match.

Cute comments that say "# Here is a workaround because awk is dumb" are passed through unmodified.

Awk scripts are often embedded in a shell script that pipes stuff into and out of awk. Often the shell script wrapper can be incorporated into the perl script, since perl can start up pipes into and out of itself, and can do other things that awk can't do by itself.

Scripts that refer to the special variables RSTART and RLENGTH can often be simplified by referring to the variables $`, $& and $', as long as they are within the scope of the pattern match that sets them.

The produced perl script may have subroutines defined to deal with awk's semantics regarding getline and print. Since a2p usually picks correctness over efficiency. it is almost always possible to rewrite such code to be more efficient by discarding the semantic sugar.

For efficiency, you may wish to remove the keyword from any return statement that is the last statement executed in a subroutine. A2p catches the most common case, but doesn't analyze embedded blocks for subtler cases.

ARGV[0] translates to `$ARGV0`, but ARGV[n] translates to `$ARGV`[$n]. A loop that tries to iterate over ARGV[0] won't find it.

**ENVIRONMENT**

A2p uses no environment variables.

**AUTHOR**

Larry Wall <*larry@wall.org*>

**FILES**

**SEE ALSO**

```
perl   The perl compiler/interpreter

s2p    sed to perl translator
```

**DIAGNOSTICS**

**BUGS**

It would be possible to emulate awk's behavior in selecting string versus numeric operations at run time by inspection of the operands, but it would be gross and inefficient. Besides, a2p almost always guesses right.

Storage for the awk syntax tree is currently static, and can run out.

**NAME**

c2ph, pstruct – Dump C structures as generated from `cc -g -S` stabs

**SYNOPSIS**

```
c2ph [-dpnP] [var=val] [files ...]
```

**OPTIONS**

```
Options:

-w  wide; short for: type_width=45 member_width=35 offset_width=8
-x  hex; short for:  offset_fmt=x offset_width=08 size_fmt=x size_width=04

-n  do not generate perl code  (default when invoked as pstruct)
-p  generate perl code         (default when invoked as c2ph)
-v  generate perl code, with C decls as comments

-i  do NOT recompute sizes for intrinsic datatypes
-a  dump information on intrinsics also

-t  trace execution
-d  spew reams of debugging output

-slist  give comma-separated list a structures to dump
```

**DESCRIPTION**

The following is the old c2ph.doc documentation by Tom Christiansen <tchrist@perl.com> Date: 25 Jul 91 08:10:21 GMT

Once upon a time, I wrote a program called pstruct. It was a perl program that tried to parse out C structures and display their member offsets for you. This was especially useful for people looking at binary dumps or poking around the kernel.

Pstruct was not a pretty program. Neither was it particularly robust. The problem, you see, was that the C compiler was much better at parsing C than I could ever hope to be.

So I got smart: I decided to be lazy and let the C compiler parse the C, which would spit out debugger stabs for me to read. These were much easier to parse. It's still not a pretty program, but at least it's more robust.

Pstruct takes any .c or .h files, or preferably .s ones, since that's the format it is going to massage them into anyway, and spits out listings like this:

```
struct tty {
  int                          tty.t_locker                   000      4
  int                          tty.t_mutex_index              004      4
  struct tty *                 tty.t_tp_virt                  008      4
  struct clist                 tty.t_rawq                     00c     20
    int                        tty.t_rawq.c_cc                00c      4
    int                        tty.t_rawq.c_cmax              010      4
    int                        tty.t_rawq.c_cfx               014      4
    int                        tty.t_rawq.c_clx               018      4
    struct tty *               tty.t_rawq.c_tp_cpu            01c      4
    struct tty *               tty.t_rawq.c_tp_iop            020      4
    unsigned char *            tty.t_rawq.c_buf_cpu           024      4
    unsigned char *            tty.t_rawq.c_buf_iop           028      4
  struct clist                 tty.t_canq                     02c     20
    int                        tty.t_canq.c_cc                02c      4
    int                        tty.t_canq.c_cmax              030      4
    int                        tty.t_canq.c_cfx               034      4
    int                        tty.t_canq.c_clx               038      4
    struct tty *               tty.t_canq.c_tp_cpu            03c      4
    struct tty *               tty.t_canq.c_tp_iop            040      4
    unsigned char *            tty.t_canq.c_buf_cpu           044      4
    unsigned char *            tty.t_canq.c_buf_iop           048      4
  struct clist                 tty.t_outq                     04c     20
    int                        tty.t_outq.c_cc                04c      4
    int                        tty.t_outq.c_cmax              050      4
    int                        tty.t_outq.c_cfx               054      4
    int                        tty.t_outq.c_clx               058      4
    struct tty *               tty.t_outq.c_tp_cpu            05c      4
    struct tty *               tty.t_outq.c_tp_iop            060      4
    unsigned char *            tty.t_outq.c_buf_cpu           064      4
    unsigned char *            tty.t_outq.c_buf_iop           068      4
  (*int)()                     tty.t_oproc_cpu                06c      4
  (*int)()                     tty.t_oproc_iop                070      4
  (*int)()                     tty.t_stopproc_cpu             074      4
  (*int)()                     tty.t_stopproc_iop             078      4
  struct thread *              tty.t_rsel                     07c      4
```

etc.

Actually, this was generated by a particular set of options. You can control the formatting of each column, whether you prefer wide or fat, hex or decimal, leading zeroes or whatever.

All you need to be able to use this is a C compiler than generates BSD/GCC−style stabs. The **−g** option on native BSD compilers and GCC should get this for you.

To learn more, just type a bogus option, like **−\?**, and a long usage message will be provided. There are a fair number of possibilities.

If you're only a C programmer, than this is the end of the message for you. You can quit right now, and if you care to, save off the source and run it when you feel like it. Or not.

But if you're a perl programmer, then for you I have something much more wondrous than just a structure offset printer.

You see, if you call pstruct by its other incybernation, c2ph, you have a code generator that translates C code into perl code! Well, structure and union declarations at least, but that's quite a bit.

Prior to this point, anyone programming in perl who wanted to interact with C programs, like the kernel, was forced to guess the layouts of the C strutures, and then hardwire these into his program. Of course, when you took your wonderfully crafted program to a system where the sgtty structure was laid out differently, you program broke. Which is a shame.

We've had Larry's h2ph translator, which helped, but that only works on cpp symbols, not real C, which was also very much needed. What I offer you is a symbolic way of getting at all the C structures. I've couched them in terms of packages and functions. Consider the following program:

```
#!/usr/local/bin/perl

require 'syscall.ph';
require 'sys/time.ph';
require 'sys/resource.ph';

$ru = "\0" x &rusage'sizeof();

syscall(&SYS_getrusage, &RUSAGE_SELF, $ru)      && die "getrusage: $!";

@ru = unpack($t = &rusage'typedef(), $ru);

$utime =  $ru[ &rusage'ru_utime + &timeval'tv_sec  ]
       + ($ru[ &rusage'ru_utime + &timeval'tv_usec ]) / 1e6;

$stime =  $ru[ &rusage'ru_stime + &timeval'tv_sec  ]
       + ($ru[ &rusage'ru_stime + &timeval'tv_usec ]) / 1e6;

printf "you have used %8.3fs+%8.3fu seconds.\n", $utime, $stime;
```

As you see, the name of the package is the name of the structure. Regular fields are just their own names. Plus the following accessor functions are provided for your convenience:

```
struct      This takes no arguments, and is merely the number of first-level
            elements in the structure.  You would use this for indexing
            into arrays of structures, perhaps like this

                $usec = $u[ &user'u_utimer
                            + (&ITIMER_VIRTUAL * &itimerval'struct)
                            + &itimerval'it_value
                            + &timeval'tv_usec
                          ];

sizeof      Returns the bytes in the structure, or the member if
            you pass it an argument, such as

                    &rusage'sizeof(&rusage'ru_utime)

typedef     This is the perl format definition for passing to pack and
            unpack.  If you ask for the typedef of a nothing, you get
            the whole structure, otherwise you get that of the member
            you ask for.  Padding is taken care of, as is the magic to
            guarantee that a union is unpacked into all its aliases.
            Bitfields are not quite yet supported however.
```

```
offsetof    This function is the byte offset into the array of that
            member.  You may wish to use this for indexing directly
            into the packed structure with vec() if you're too lazy
            to unpack it.

typeof      Not to be confused with the typedef accessor function, this
            one returns the C type of that field.  This would allow
            you to print out a nice structured pretty print of some
            structure without knoning anything about it beforehand.
            No args to this one is a noop.  Someday I'll post such
            a thing to dump out your u structure for you.
```

The way I see this being used is like basically this:

```
% h2ph <some_include_file.h  >  /usr/lib/perl/tmp.ph
% c2ph  some_include_file.h  >> /usr/lib/perl/tmp.ph
% install
```

It's a little tricker with c2ph because you have to get the includes right. I can't know this for your system, but it's not usually too terribly difficult.

The code isn't pretty as I mentioned -- I never thought it would be a 1000- line program when I started, or I might not have begun. :–) But I would have been less cavalier in how the parts of the program communicated with each other, etc. It might also have helped if I didn't have to divine the makeup of the stabs on the fly, and then account for micro differences between my compiler and gcc.

Anyway, here it is. Should run on perl v4 or greater. Maybe less.

```
--tom
```

## NAME

h2ph − convert .h C header files to .ph Perl header files

## SYNOPSIS

**h2ph [−d destination directory] [−r | −a] [−l] [headerfiles]**

## DESCRIPTION

*h2ph* converts any C header files specified to the corresponding Perl header file format. It is most easily run while in /usr/include:

```
cd /usr/include; h2ph * sys/*
```

or

```
cd /usr/include; h2ph -r -l .
```

The output files are placed in the hierarchy rooted at Perl's architecture dependent library directory. You can specify a different hierarchy with a **−d** switch.

If run with no arguments, filters standard input to standard output.

## OPTIONS

-d destination_dir
: Put the resulting **.ph** files beneath **destination_dir**, instead of beneath the default Perl library location (`$Config{'installsitsearch'}`).

-r
: Run recursively; if any of **headerfiles** are directories, then run *h2ph* on all files in those directories (and their subdirectories, etc.). **−r** and **−a** are mutually exclusive.

-a
: Run automagically; convert **headerfiles**, as well as any **.h** files which they include. This option will search for **.h** files in all directories which your C compiler ordinarily uses. **−a** and **−r** are mutually exclusive.

-l
: Symbolic links will be replicated in the destination directory. If **−l** is not specified, then links are skipped over.

-h
: Put "hints" in the .ph files which will help in locating problems with *h2ph*. In those cases when you **require** a **.ph** file containing syntax errors, instead of the cryptic

```
[ some error condition ] at (eval mmm) line nnn
```

you will see the slightly more helpful

```
[ some error condition ] at filename.ph line nnn
```

However, the **.ph** files almost double in size when built using **−h**.

-D
: Include the code from the **.h** file as a comment in the **.ph** file. This is primarily used for debugging *h2ph*.

## ENVIRONMENT

No environment variables are used.

## FILES

```
/usr/include/*.h
/usr/include/sys/*.h
```

etc.

## AUTHOR

Larry Wall

**SEE ALSO**

*perl*(1)

**DIAGNOSTICS**

The usual warnings if it can't read or write the files involved.

**BUGS**

Doesn't construct the `%sizeof` array for you.

It doesn't handle all C constructs, but it does attempt to isolate definitions inside evals so that you can get at the definitions that it can translate.

It's only intended as a rough tool. You may need to dicker with the files produced.

**NAME**

h2xs − convert .h C header files to Perl extensions

**SYNOPSIS**

**h2xs** [**−AOPXcdf**] [**−v** version] [**−n** module_name] [**−p** prefix] [**−s** sub] [headerfile ... [extra_libraries]]

**h2xs −h**

**DESCRIPTION**

*h2xs* builds a Perl extension from C header files. The extension will include functions which can be used to retrieve the value of any #define statement which was in the C header files.

The *module_name* will be used for the name of the extension. If module_name is not supplied then the name of the first header file will be used, with the first character capitalized.

If the extension might need extra libraries, they should be included here. The extension Makefile.PL will take care of checking whether the libraries actually exist and how they should be loaded. The extra libraries should be specified in the form −lm −lposix, etc, just as on the cc command line. By default, the Makefile.PL will search through the library path determined by Configure. That path can be augmented by including arguments of the form **−L/another/library/path** in the extra-libraries argument.

**OPTIONS**

**−A** Omit all autoload facilities. This is the same as **−c** but also removes the `require AutoLoader` statement from the .pm file.

**−F** Additional flags to specify to C preprocessor when scanning header for function declarations. Should not be used without **−x**.

**−O** Allows a pre-existing extension directory to be overwritten.

**−P** Omit the autogenerated stub POD section.

**−X** Omit the XS portion. Used to generate templates for a module which is not XS−based.

**−c** Omit `constant()` from the .xs file and corresponding specialised `AUTOLOAD` from the .pm file.

**−d** Turn on debugging messages.

**−f** Allows an extension to be created for a header even if that header is not found in /usr/include.

**−h** Print the usage, help and version for this h2xs and exit.

**−n** *module_name*

Specifies a name to be used for the extension, e.g., −n RPC::DCE

**−p** *prefix*

Specify a prefix which should be removed from the Perl function names, e.g., −p sec_rgy_ This sets up the XS **PREFIX** keyword and removes the prefix from functions that are autoloaded via the `constant()` mechansim.

**−s** *sub1,sub2*

Create a perl subroutine for the specified macros rather than autoload with the *constant()* subroutine. These macros are assumed to have a return type of **char ***, e.g., −s sec_rgy_wildcard_name,sec_rgy_wildcard_sid.

**−v** *version*

Specify a version number for this extension. This version number is added to the templates. The default is 0.01.

**−x** Automatically generate XSUBs basing on function declarations in the header file. The package `C::Scan` should be installed. If this option is specified, the name of the header file may look like `NAME1,NAME2`. In this case NAME1 is used instead of the specified string, but XSUBs are emitted only for the declarations included from file NAME2.

Note that some types of arguments/return-values for functions may result in XSUB−declarations/typemap-entries which need hand-editing. Such may be objects which cannot be converted from/to a pointer (like `long long`), pointers to functions, or arrays.

**EXAMPLES**

```
# Default behavior, extension is Rusers
h2xs rpcsvc/rusers

# Same, but extension is RUSERS
h2xs -n RUSERS rpcsvc/rusers

# Extension is rpcsvc::rusers. Still finds <rpcsvc/rusers.h>
h2xs rpcsvc::rusers

# Extension is ONC::RPC.  Still finds <rpcsvc/rusers.h>
h2xs -n ONC::RPC rpcsvc/rusers

# Without constant() or AUTOLOAD
h2xs -c rpcsvc/rusers

# Creates templates for an extension named RPC
h2xs -cfn RPC

# Extension is ONC::RPC.
h2xs -cfn ONC::RPC

# Makefile.PL will look for library -lrpc in
# additional directory /opt/net/lib
h2xs rpcsvc/rusers -L/opt/net/lib -lrpc

# Extension is DCE::rgynbase
# prefix "sec_rgy_" is dropped from perl function names
h2xs -n DCE::rgynbase -p sec_rgy_ dce/rgynbase

# Extension is DCE::rgynbase
# prefix "sec_rgy_" is dropped from perl function names
# subroutines are created for sec_rgy_wildcard_name and sec_rgy_wildcard_sid
h2xs -n DCE::rgynbase -p sec_rgy_ \
-s sec_rgy_wildcard_name,sec_rgy_wildcard_sid dce/rgynbase

# Make XS without defines in perl.h, but with function declarations
# visible from perl.h. Name of the extension is perl1.
# When scanning perl.h, define -DEXT=extern -DdEXT= -DINIT(x)=
# Extra backslashes below because the string is passed to shell.
# Note that a directory with perl header files would
#  be added automatically to include path.
h2xs -xAn perl1 -F "-DEXT=extern -DdEXT= -DINIT\(x\)=" perl.h

# Same with function declaration in proto.h as visible from perl.h.
h2xs -xAn perl2 perl.h,proto.h
```

**ENVIRONMENT**

No environment variables are used.

**AUTHOR**

Larry Wall and others

**SEE ALSO**

the *perl* manpage, the *perlxstut* manpage, the *ExtUtils::MakeMaker* manpage, and the *AutoLoader* manpage.

**DIAGNOSTICS**

The usual warnings if it cannot read or write the files involved.

## NAME

perlbug – how to submit bug reports on Perl

## SYNOPSIS

**perlbug** [ **–v** ] [ **–a** *address* ] [ **–s** *subject* ] [ **–b** *body* | **–f** *inputfile* ] [ **–F** *outputfile* ] [ **–r** *returnaddress* ] [ **–e** *editor* ] [ **–c** *adminaddress* | **–C** ] [ **–S** ] [ **–t** ] [ **–d** ] [ **–h** ]

**perlbug** [ **–v** ] [ **–r** *returnaddress* ] [ **–ok** | **–okay** | **–nok** | **–nokay** ]

## DESCRIPTION

A program to help generate bug reports about perl or the modules that come with it, and mail them.

If you have found a bug with a non-standard port (one that was not part of the *standard distribution*), a binary distribution, or a non-standard module (such as Tk, CGI, etc), then please see the documentation that came with that distribution to determine the correct place to report bugs.

`perlbug` is designed to be used interactively. Normally no arguments will be needed. Simply run it, and follow the prompts.

If you are unable to run **perlbug** (most likely because you don't have a working setup to send mail that perlbug recognizes), you may have to compose your own report, and email it to **perlbug@perl.com**. You might find the **–d** option useful to get summary information in that case.

In any case, when reporting a bug, please make sure you have run through this checklist:

What version of perl you are running?

Type `perl -v` at the command line to find out.

Are you running the latest released version of perl?

Look at http://www.perl.com/ to find out. If it is not the latest released version, get that one and see whether your bug has been fixed. Note that bug reports about old versions of perl, especially those prior to the 5.0 release, are likely to fall upon deaf ears. You are on your own if you continue to use perl1 .. perl4.

Are you sure what you have is a bug?

A significant number of the bug reports we get turn out to be documented features in perl. Make sure the behavior you are witnessing doesn't fall under that category, by glancing through the documentation that comes with perl (we'll admit this is no mean task, given the sheer volume of it all, but at least have a look at the sections that *seem* relevant).

Be aware of the familiar traps that perl programmers of various hues fall into. See the *perltrap* manpage.

Try to study the problem under the perl debugger, if necessary. See the *perldebug* manpage.

Do you have a proper test case?

The easier it is to reproduce your bug, the more likely it will be fixed, because if no one can duplicate the problem, no one can fix it. A good test case has most of these attributes: fewest possible number of lines; few dependencies on external commands, modules, or libraries; runs on most platforms unimpeded; and is self-documenting.

A good test case is almost always a good candidate to be on the perl test suite. If you have the time, consider making your test case so that it will readily fit into the standard test suite.

Can you describe the bug in plain English?

The easier it is to understand a reproducible bug, the more likely it will be fixed. Anything you can provide by way of insight into the problem helps a great deal. In other words, try to analyse the problem to the extent you feel qualified and report your discoveries.

Can you fix the bug yourself?

A bug report which *includes a patch to fix it* will almost definitely be fixed. Use the `diff` program to generate your patches (`diff` is being maintained by the GNU folks as part of the **diffutils** package, so you should be able to get it from any of the GNU software repositories). If you do submit a patch, the cool-dude counter at perlbug@perl.com will register you as a savior of the world. Your patch may be returned with requests for changes, or requests for more detailed explanations about your fix.

Here are some clues for creating quality patches: Use the **–c** or **–u** switches to the diff program (to create a so-called context or unified diff). Make sure the patch is not reversed (the first argument to diff is typically the original file, the second argument your changed file). Make sure you test your patch by applying it with the `patch` program before you send it on its way. Try to follow the same style as the code you are trying to patch. Make sure your patch really does work (`make test`, if the thing you're patching supports it).

Can you use `perlbug` to submit the report?
: **perlbug** will, amongst other things, ensure your report includes crucial information about your version of perl. If `perlbug` is unable to mail your report after you have typed it in, you may have to compose the message yourself, add the output produced by `perlbug -d` and email it to **perlbug@perl.com**. If, for some reason, you cannot run `perlbug` at all on your system, be sure to include the entire output produced by running `perl -V` (note the uppercase V).

Having done your bit, please be prepared to wait, to be told the bug is in your code, or even to get no reply at all. The perl maintainers are busy folks, so if your problem is a small one or if it is difficult to understand or already known, they may not respond with a personal reply. If it is important to you that your bug be fixed, do monitor the `Changes` file in any development releases since the time you submitted the bug, and encourage the maintainers with kind words (but never any flames!). Feel free to resend your bug report if the next released version of perl comes out and your bug is still present.

## OPTIONS

**–a** Address to send the report to. Defaults to 'perlbug@perl.com'.

**–b** Body of the report. If not included on the command line, or in a file with **–f**, you will get a chance to edit the message.

**–C** Don't send copy to administrator.

**–c** Address to send copy of report to. Defaults to the address of the local perl administrator (recorded when perl was built).

**–d** Data mode (the default if you redirect or pipe output). This prints out your configuration data, without mailing anything. You can use this with **–v** to get more complete data.

**–e** Editor to use.

**–f** File containing the body of the report. Use this to quickly send a prepared message.

**–F** File to output the results to instead of sending as an email. Useful particularly when running perlbug on a machine with no direct internet connection.

**–h** Prints a brief summary of the options.

**–ok** Report successful build on this system to perl porters. Forces **–S** and **–C**. Forces and supplies values for **–s** and **–b**. Only prompts for a return address if it cannot guess it (for use with **make**). Honors return address specified with **–r**. You can use this with **–v** to get more complete data. Only makes a report if this system is less than 60 days old.

**–okay** As **–ok** except it will report on older systems.

**–nok** Report unsuccessful build on this system. Forces **–C**. Forces and supplies a value for **–s**, then requires you to edit the report and say what went wrong. Alternatively, a prepared report may be supplied using **–f**. Only prompts for a return address if it cannot guess it (for use with **make**). Honors return address specified with **–r**. You can use this with **–v** to get more complete data. Only makes a report if this system is less than 60 days old.

**–nokay** As **–nok** except it will report on older systems.

**–r** Your return address. The program will ask you to confirm its default if you don't use this option.

**−S** Send without asking for confirmation.

**−s** Subject to include with the message. You will be prompted if you don't supply one on the command line.

**−t** Test mode. The target address defaults to 'perlbug-test@perl.com'.

**−v** Include verbose configuration data in the report.

**AUTHORS**

Kenneth Albanowski (<kjahds@kjahds.com>), subsequently *doc*tored by Gurusamy Sarathy (<gsar@umich.edu>), Tom Christiansen (<tchrist@perl.com>), Nathan Torkington (<gnat@frii.com>), Charles F. Randall (<cfr@pobox.com>), Mike Guy (<mjtg@cam.a.uk>), Dominic Dunlop (<domo@computer.org>) and Hugo van der Sanden (<hv@crypt0.demon.co.uk>).

**SEE ALSO**

*perl* (1), *perldebug* (1), *perltrap* (1), *diff* (1), *patch* (1)

**BUGS**

None known (guess what must have been used to report them?)

**NAME**

perldoc – Look up Perl documentation in pod format.

**SYNOPSIS**

**perldoc** [**–h**] [**–v**] [**–t**] [**–u**] [**–m**] [**–l**] [**–F**] [**–X**] PageName|ModuleName|ProgramName

**perldoc –f** BuiltinFunction

**perldoc –q** FAQ Keyword

**DESCRIPTION**

*perldoc* looks up a piece of documentation in .pod format that is embedded in the perl installation tree or in a perl script, and displays it via `pod2man | nroff -man | $PAGER`. (In addition, if running under HP–UX, `col -x` will be used.) This is primarily used for the documentation for the perl library modules.

Your system may also have man pages installed for those modules, in which case you can probably just use the *man* (1) command.

**OPTIONS**

**–h** help
: Prints out a brief help message.

**–v** verbose
: Describes search for the item in detail.

**–t** text output
: Display docs using plain text converter, instead of nroff. This may be faster, but it won't look as nice.

**–u** unformatted
: Find docs only; skip reformatting by pod2*

**–m** module
: Display the entire module: both code and unformatted pod documentation. This may be useful if the docs don't explain a function in the detail you need, and you'd like to inspect the code directly; perldoc will find the file for you and simply hand it off for display.

**–l** file name only
: Display the file name of the module found.

**–F** file names
: Consider arguments as file names, no search in directories will be performed.

**–f** perlfunc
: The **–f** option followed by the name of a perl built in function will extract the documentation of this function from the *perlfunc* manpage.

**–q** perlfaq
: The **–q** option takes a regular expression as an argument. It will search the question headings in perlfaq[1-9] and print the entries matching the regular expression.

**–X** use an index if present
: The **–X** option looks for a entry whose basename matches the name given on the command line in the file `$Config{archlib}/pod.idx`. The pod.idx file should contain fully qualified filenames, one per line.

**PageName|ModuleName|ProgramName**
: The item you want to look up. Nested modules (such as `File::Basename`) are specified either as `File::Basename` or `File/Basename`. You may also give a descriptive name of a page, such as `perlfunc`. You make also give a partial or wrong-case name, such as "basename" for "File::Basename", but this will be slower, if there is more then one page with the same partial name, you will only get the first one.

**ENVIRONMENT**

Any switches in the `PERLDOC` environment variable will be used before the command line arguments. `perldoc` also searches directories specified by the `PERL5LIB` (or `PERLLIB` if `PERL5LIB` is not defined) and `PATH` environment variables. (The latter is so that embedded pods for executables, such as `perldoc` itself, are available.) `perldoc` will use, in order of preference, the pager defined in `PERLDOC_PAGER`, `MANPAGER`, or `PAGER` before trying to find a pager on its own. (`MANPAGER` is not used if `perldoc` was told to display plain text or unformatted pod.)

**AUTHOR**

Kenneth Albanowski <kjahds@kjahds.com>

Minor updates by Andy Dougherty <doughera@lafcol.lafayette.edu>

### NAME

pl2pm − Rough tool to translate Perl4 .pl files to Perl5 .pm modules.

### SYNOPSIS

**pl2pm** *files*

### DESCRIPTION

**pl2pm** is a tool to aid in the conversion of Perl4-style .pl library files to Perl5-style library modules. Usually, your old .pl file will still work fine and you should only use this tool if you plan to update your library to use some of the newer Perl 5 features, such as AutoLoading.

### LIMITATIONS

It's just a first step, but it's usually a good first step.

### AUTHOR

Larry Wall <larry@wall.org>

**NAME**

pod2html – convert .pod files to .html files

**SYNOPSIS**

```
pod2html --help --htmlroot=<name> --infile=<name> --outfile=<name>
         --podpath=<name>:...:<name> --podroot=<name>
         --libpods=<name>:...:<name> --recurse --norecurse --verbose
         --index --noindex --title=<name>
```

**DESCRIPTION**

Converts files from pod format (see the *perlpod* manpage) to HTML format.

**ARGUMENTS**

pod2html takes the following arguments:

help

```
--help
```

Displays the usage message.

htmlroot

```
--htmlroot=name
```

Sets the base URL for the HTML files. When cross-references are made, the HTML root is prepended to the URL.

infile

```
--infile=name
```

Specify the pod file to convert. Input is taken from STDIN if no infile is specified.

outfile

```
--outfile=name
```

Specify the HTML file to create. Output goes to STDOUT if no outfile is specified.

podroot

```
--podroot=name
```

Specify the base directory for finding library pods.

podpath

```
--podpath=name:...:name
```

Specify which subdirectories of the podroot contain pod files whose HTML converted forms can be linked-to in cross-references.

libpods

```
--libpods=name:...:name
```

List of page names (eg, "perlfunc") which contain linkable `=items`.

netscape

```
--netscape
```

Use Netscape HTML directives when applicable.

nonetscape

```
--nonetscape
```

Do not use Netscape HTML directives (default).

index

```
--index
```

Generate an index at the top of the HTML file (default behaviour).

noindex

```
--noindex
```

Do not generate an index at the top of the HTML file.

recurse

```
--recurse
```

Recurse into subdirectories specified in podpath (default behaviour).

norecurse

```
--norecurse
```

Do not recurse into subdirectories specified in podpath.

title

```
--title=title
```

Specify the title of the resulting HTML file.

verbose

```
--verbose
```

Display progress messages.

**AUTHOR**

Tom Christiansen, <tchrist@perl.com>.

**BUGS**

See the *Pod::Html* manpage for a list of known bugs in the translator.

**SEE ALSO**

the *perlpod* manpage, the *Pod::HTML* manpage

**COPYRIGHT**

This program is distributed under the Artistic License.

**NAME**

pod2man − translate embedded Perl pod directives into man pages

**SYNOPSIS**

**pod2man** [ **--section=***manext* ] [ **--release=***relpatch* ] [ **--center=***string* ] [ **--date=***string* ] [ **--fixed=***font* ] [ **--official** ] [ **--lax** ] *inputfile*

**DESCRIPTION**

**pod2man** converts its input file containing embedded pod directives (see the *perlpod* manpage) into nroff source suitable for viewing with *nroff*(1) or *troff*(1) using the *man*(7) macro set.

Besides the obvious pod conversions, **pod2man** also takes care of *func()*, *func*(n), and simple variable references like `$foo` or `@bar` so you don't have to use code escapes for them; complex expressions like `$fred{'stuff'}` will still need to be escaped, though. Other nagging little roffish things that it catches include translating the minus in something like foo-bar, making a long dash— like this— into a real em dash, fixing up ''paired quotes'', putting a little space after the parens in something like *func()*, making C++ and $\pi$ look right, making double underbars have a little tiny space between them, making ALLCAPS a teeny bit smaller in *troff*(1), and escaping backslashes so you don't have to.

**OPTIONS**

center
: Set the centered header to a specific string. The default is ''User Contributed Perl Documentation'', unless the `--official` flag is given, in which case the default is ''Perl Programmers Reference Guide''.

date
: Set the left-hand footer string to this value. By default, the modification date of the input file will be used.

fixed
: The fixed font to use for code refs. Defaults to CW.

official
: Set the default header to indicate that this page is of the standard release in case `--center` is not given.

release
: Set the centered footer. By default, this is the current perl release.

section
: Set the section for the `.TH` macro. The standard conventions on sections are to use 1 for user commands, 2 for system calls, 3 for functions, 4 for devices, 5 for file formats, 6 for games, 7 for miscellaneous information, and 8 for administrator commands. This works best if you put your Perl man pages in a separate tree, like */usr/local/perl/man/*. By default, section 1 will be used unless the file ends in *.pm* in which case section 3 will be selected.

lax
: Don't complain when required sections aren't present.

**Anatomy of a Proper Man Page**

For those not sure of the proper layout of a man page, here's an example of the skeleton of a proper man page. Head of the major headers should be setout as a `=head1` directive, and are historically written in the rather startling ALL UPPER CASE format, although this is not mandatory. Minor headers may be included using `=head2`, and are typically in mixed case.

NAME
: Mandatory section; should be a comma-separated list of programs or functions documented by this podpage, such as:

```
foo, bar - programs to do something
```

SYNOPSIS
: A short usage summary for programs and functions, which may someday be deemed mandatory.

DESCRIPTION
: Long drawn out discussion of the program. It's a good idea to break this up into subsections using the `=head2` directives, like

```
=head2 A Sample Subection
```

```
=head2 Yet Another Sample Subection
```

OPTIONS Some people make this separate from the description.

RETURN VALUE
What the program or function returns if successful.

ERRORS Exceptions, return codes, exit stati, and errno settings.

EXAMPLES
Give some example uses of the program.

ENVIRONMENT
Envariables this program might care about.

FILES All files used by the program. You should probably use the F<> for these.

SEE ALSO Other man pages to check out, like *man* (1), *man* (7), *makewhatis* (8), or *catman* (8).

NOTES Miscellaneous commentary.

CAVEATS Things to take special care with; sometimes called WARNINGS.

DIAGNOSTICS
All possible messages the program can print out—and what they mean.

BUGS Things that are broken or just don't work quite right.

RESTRICTIONS
Bugs you don't plan to fix :–)

AUTHOR Who wrote it (or AUTHORS if multiple).

HISTORY Programs derived from other sources sometimes have this, or you might keep a modification log here.

**EXAMPLES**

```
pod2man program > program.1
pod2man some_module.pm > /usr/perl/man/man3/some_module.3
pod2man --section=7 note.pod > note.7
```

**DIAGNOSTICS**

The following diagnostics are generated by **pod2man**. Items marked "(W)" are non-fatal, whereas the "(F)" errors will cause **pod2man** to immediately exit with a non-zero status.

bad option in paragraph %d of %s: "%s" should be [%s]<%s>
(W) If you start include an option, you should set it off as bold, italic, or code.

can't open %s: %s
(F) The input file wasn't available for the given reason.

Improper man page – no dash in NAME header in paragraph %d of %s
(W) The NAME header did not have an isolated dash in it. This is considered important.

Invalid man page – no NAME line in %s
(F) You did not include a NAME header, which is essential.

roff font should be 1 or 2 chars, not '%s' (F)
(F) The font specified with the `--fixed` option was not a one- or two-digit roff font.

%s is missing required section: %s
(W) Required sections include NAME, DESCRIPTION, and if you're using a section starting with a 3, also a SYNOPSIS. Actually, not having a NAME is a fatal.

Unknown escape: %s in %s

(W) An unknown HTML entity (probably for an 8-bit character) was given via a `E<>` directive. Besides amp, lt, gt, and quot, recognized entities are Aacute, aacute, Acirc, acirc, AElig, aelig, Agrave, agrave, Aring, aring, Atilde, atilde, Auml, auml, Ccedil, ccedil, Eacute, eacute, Ecirc, ecirc, Egrave, egrave, ETH, eth, Euml, euml, Iacute, iacute, Icirc, icirc, Igrave, igrave, Iuml, iuml, Ntilde, ntilde, Oacute, oacute, Ocirc, ocirc, Ograve, ograve, Oslash, oslash, Otilde, otilde, Ouml, ouml, szlig, THORN, thorn, Uacute, uacute, Ucirc, ucirc, Ugrave, ugrave, Uuml, uuml, Yacute, yacute, and yuml.

Unmatched =back

(W) You have a `=back` without a corresponding `=over`.

Unrecognized pod directive: %s

(W) You specified a pod directive that isn't in the known list of `=head1`, `=head2`, `=item`, `=over`, `=back`, or `=cut`.

**NOTES**

If you would like to print out a lot of man page continuously, you probably want to set the C and D registers to set contiguous page numbering and even/odd paging, at least on some versions of *man* (7). Settting the F register will get you some additional experimental indexing:

```
troff -man -rC1 -rD1 -rF1 perl.1 perldata.1 perlsyn.1 ...
```

The indexing merely outputs messages via `.tm` for each major page, section, subsection, item, and any `X<>` directives.

**RESTRICTIONS**

None at this time.

**BUGS**

The =over and =back directives don't really work right. They take absolute positions instead of offsets, don't nest well, and making people count is suboptimal in any event.

**AUTHORS**

Original prototype by Larry Wall, but so massively hacked over by Tom Christiansen such that Larry probably doesn't recognize it anymore.

## NAME

s2p − Sed to Perl translator

## SYNOPSIS

**s2p [options] filename**

## DESCRIPTION

*S2p* takes a sed script specified on the command line (or from standard input) and produces a comparable *perl* script on the standard output.

### Options

Options include:

**−D<number>**
sets debugging flags.

**−n** specifies that this sed script was always invoked with a **sed −n**. Otherwise a switch parser is prepended to the front of the script.

**−p** specifies that this sed script was never invoked with a **sed −n**. Otherwise a switch parser is prepended to the front of the script.

### Considerations

The perl script produced looks very sed-ish, and there may very well be better ways to express what you want to do in perl. For instance, s2p does not make any use of the split operator, but you might want to.

The perl script you end up with may be either faster or slower than the original sed script. If you're only interested in speed you'll just have to try it both ways. Of course, if you want to do something sed doesn't do, you have no choice. It's often possible to speed up the perl script by various methods, such as deleting all references to $\ and chop.

## ENVIRONMENT

S2p uses no environment variables.

## AUTHOR

Larry Wall <*larry@wall.org*>

## FILES

## SEE ALSO

```
perl   The perl compiler/interpreter

a2p    awk to perl translator
```

## DIAGNOSTICS

## BUGS

**NAME**

diagnostics − Perl compiler pragma to force verbose warning diagnostics

splain − standalone program to do the same thing

**SYNOPSIS**

As a pragma:

```
use diagnostics;
use diagnostics -verbose;

enable  diagnostics;
disable diagnostics;
```

Aa a program:

```
perl program 2>diag.out
splain [-v] [-p] diag.out
```

**DESCRIPTION**

**The** `diagnostics` Pragma

This module extends the terse diagnostics normally emitted by both the perl compiler and the perl interpeter, augmenting them with the more explicative and endearing descriptions found in the *perldiag* manpage. Like the other pragmata, it affects the compilation phase of your program rather than merely the execution phase.

To use in your program as a pragma, merely invoke

```
use diagnostics;
```

at the start (or near the start) of your program. (Note that this *does* enable perl's **−w** flag.) Your whole compilation will then be *subject* (ed :−) to the enhanced diagnostics. These still go out **STDERR**.

Due to the interaction between runtime and compiletime issues, and because it's probably not a very good idea anyway, you may not use `no diagnostics` to turn them off at compiletime. However, you may control there behaviour at runtime using the *disable()* and *enable()* methods to turn them off and on respectively.

The **−verbose** flag first prints out the the *perldiag* manpage introduction before any other diagnostics. The `$diagnostics::PRETTY` variable can generate nicer escape sequences for pagers.

**The** *splain* Program

While apparently a whole nuther program, *splain* is actually nothing more than a link to the (executable) *diagnostics.pm* module, as well as a link to the *diagnostics.pod* documentation. The **−v** flag is like the `use diagnostics -verbose` directive. The **−p** flag is like the `$diagnostics::PRETTY` variable. Since you're post-processing with *splain*, there's no sense in being able to *enable()* or *disable()* processing.

Output from *splain* is directed to **STDOUT**, unlike the pragma.

**EXAMPLES**

The following file is certain to trigger a few errors at both runtime and compiletime:

```
use diagnostics;
print NOWHERE "nothing\n";
print STDERR "\n\tThis message should be unadorned.\n";
warn "\tThis is a user warning";
print "\nDIAGNOSTIC TESTER: Please enter a <CR> here: ";
my $a, $b = scalar <STDIN>;
print "\n";
print $x/$y;
```

If you prefer to run your program first and look at its problem afterwards, do this:

```
perl -w test.pl 2>test.out
./splain < test.out
```

Note that this is not in general possible in shells of more dubious heritage, as the theoretical

```
(perl -w test.pl >/dev/tty) >& test.out
./splain < test.out
```

Because you just moved the existing **stdout** to somewhere else.

If you don't want to modify your source code, but still have on-the-fly warnings, do this:

```
exec 3>&1; perl -w test.pl 2>&1 1>&3 3>&- | splain 1>&2 3>&-
```

Nifty, eh?

If you want to control warnings on the fly, do something like this. Make sure you do the `use` first, or you won't be able to get at the *enable()* or *disable()* methods.

```
use diagnostics; # checks entire compilation phase
    print "\ntime for 1st bogus diags: SQUAWKINGS\n";
    print BOGUS1 'nada';
    print "done with 1st bogus\n";

disable diagnostics; # only turns off runtime warnings
    print "\ntime for 2nd bogus: (squelched)\n";
    print BOGUS2 'nada';
    print "done with 2nd bogus\n";

enable diagnostics; # turns back on runtime warnings
    print "\ntime for 3rd bogus: SQUAWKINGS\n";
    print BOGUS3 'nada';
    print "done with 3rd bogus\n";

disable diagnostics;
    print "\ntime for 4th bogus: (squelched)\n";
    print BOGUS4 'nada';
    print "done with 4th bogus\n";
```

**INTERNALS**

Diagnostic messages derive from the *perldiag.pod* file when available at runtime. Otherwise, they may be embedded in the file itself when the splain package is built. See the *Makefile* for details.

If an extant `$SIG{__WARN__}` handler is discovered, it will continue to be honored, but only after the *diagnostics::splainthis()* function (the module's `$SIG{__WARN__}` interceptor) has had its way with your warnings.

There is a `$diagnostics::DEBUG` variable you may set if you're desperately curious what sorts of things are being intercepted.

```
BEGIN { $diagnostics::DEBUG = 1 }
```

### BUGS

Not being able to say ``no diagnostics'' is annoying, but may not be insurmountable.

The `-pretty` directive is called too late to affect matters. You have to do this instead, and *before* you load the module.

```
BEGIN { $diagnostics::PRETTY = 1 }
```

I could start up faster by delaying compilation until it should be needed, but this gets a ``panic: top_level'' when using the pragma form in Perl 5.001e.

While it's true that this documentation is somewhat subserious, if you use a program named *splain*, you should expect a bit of whimsy.

### AUTHOR

Tom Christiansen <*tchrist@mox.perl.com*>, 25 June 1995.

**NAME**

xsubpp − compiler to convert Perl XS code into C code

**SYNOPSIS**

**xsubpp** [**−v**] [**−C++**] [**−except**] [**−s pattern**] [**−prototypes**] [**−noversioncheck**] [**−nolinenumbers**] [**−typemap typemap**] [**−object_capi**]... file.xs

**DESCRIPTION**

*xsubpp* will compile XS code into C code by embedding the constructs necessary to let C functions manipulate Perl values and creates the glue necessary to let Perl access those functions. The compiler uses typemaps to determine how to map C function parameters and variables to Perl values.

The compiler will search for typemap files called *typemap*. It will use the following search path to find default typemaps, with the rightmost typemap taking precedence.

```
../../../typemap:../../typemap:../typemap:typemap
```

**OPTIONS**

**−C++** Adds "extern "C"" to the C code.

**−except**
Adds exception handling stubs to the C code.

**−typemap typemap**
Indicates that a user-supplied typemap should take precedence over the default typemaps. This option may be used multiple times, with the last typemap having the highest precedence.

**−v** Prints the *xsubpp* version number to standard output, then exits.

**−prototypes**
By default *xsubpp* will not automatically generate prototype code for all xsubs. This flag will enable prototypes.

**−noversioncheck**
Disables the run time test that determines if the object file (derived from the `.xs` file) and the `.pm` files have the same version number.

**−nolinenumbers**
Prevents the inclusion of '#line' directives in the output.

**−object_capi**
Compile code as C in a PERL_OBJECT environment.

back

**ENVIRONMENT**

No environment variables are used.

**AUTHOR**

Larry Wall

**MODIFICATION HISTORY**

See the file *changes.pod*.

**SEE ALSO**

*perl*(1), *perlxs*(1), *perlxstut*(1)

## NAME

attrs – set/get attributes of a subroutine

## SYNOPSIS

```
sub foo {
    use attrs qw(locked method);
    ...
}

@a = attrs::get(\&foo);
```

## DESCRIPTION

This module lets you set and get attributes for subroutines. Setting attributes takes place at compile time; trying to set invalid attribute names causes a compile-time error. Calling `attr::get` on a subroutine reference or name returns its list of attribute names. Notice that `attr::get` is not exported. Valid attributes are as follows.

method
: Indicates that the invoking subroutine is a method.

locked
: Setting this attribute is only meaningful when the subroutine or method is to be called by multiple threads. When set on a method subroutine (i.e. one marked with the **method** attribute above), perl ensures that any invocation of it implicitly locks its first argument before execution. When set on a non-method subroutine, perl ensures that a lock is taken on the subroutine itself before execution. The semantics of the lock are exactly those of one explicitly taken with the `lock` operator immediately after the subroutine is entered.

## NAME

autouse – postpone load of modules until a function is used

## SYNOPSIS

```
use autouse 'Carp' => qw(carp croak);
carp "this carp was predeclared and autoused ";
```

## DESCRIPTION

If the module `Module` is already loaded, then the declaration

```
use autouse 'Module' => qw(func1 func2($;$) Module::func3);
```

is equivalent to

```
use Module qw(func1 func2);
```

if `Module` defines *func2()* with prototype `($;$)`, and *func1()* and *func3()* have no prototypes. (At least if `Module` uses `Exporter`'s `import`, otherwise it is a fatal error.)

If the module `Module` is not loaded yet, then the above declaration declares functions *func1()* and *func2()* in the current package, and declares a function *Module::func3()*. When these functions are called, they load the package `Module` if needed, and substitute themselves with the correct definitions.

## WARNING

Using `autouse` will move important steps of your program's execution from compile time to runtime. This can

- Break the execution of your program if the module you `autoused` has some initialization which it expects to be done early.
- hide bugs in your code since important checks (like correctness of prototypes) is moved from compile time to runtime. In particular, if the prototype you specified on `autouse` line is wrong, you will not find it out until the corresponding function is executed. This will be very unfortunate for functions which are not always called (note that for such functions `autouse`ing gives biggest win, for a workaround see below).

To alleviate the second problem (partially) it is advised to write your scripts like this:

```
use Module;
use autouse Module => qw(carp($) croak(&$));
carp "this carp was predeclared and autoused ";
```

The first line ensures that the errors in your argument specification are found early. When you ship your application you should comment out the first line, since it makes the second one useless.

## AUTHOR

Ilya Zakharevich (ilya@math.ohio-state.edu)

## SEE ALSO

*perl*(1).

## NAME

base − Establish IS−A relationship with base class at compile time

## SYNOPSIS

```
package Baz;
use base qw(Foo Bar);
```

## DESCRIPTION

Roughly similar in effect to

```
BEGIN {
    require Foo;
    require Bar;
    push @ISA, qw(Foo Bar);
}
```

Will also initialize the `%FIELDS` hash if one of the base classes has it. Multiple inheritance of `%FIELDS` is not supported. The 'base' pragma will croak if multiple base classes has a `%FIELDS` hash. See the *fields* manpage for a description of this feature.

When strict 'vars' is in scope *base* also let you assign to `@ISA` without having to declare `@ISA` with the 'vars' pragma first.

This module was introduced with Perl 5.004_04.

## SEE ALSO

the *fields* manpage

## NAME

blib – Use MakeMaker's uninstalled version of a package

## SYNOPSIS

```
perl -Mblib script [args...]

perl -Mblib=dir script [args...]
```

## DESCRIPTION

Looks for MakeMaker-like *'blib'* directory structure starting in *dir* (or current directory) and working back up to five levels of '..'.

Intended for use on command line with **–M** option as a way of testing arbitary scripts against an uninstalled version of a package.

However it is possible to :

```
use blib;
or
use blib '..';
```

etc. if you really must.

## BUGS

Pollutes global name space for development only task.

## AUTHOR

Nick Ing-Simmons nik@tiuk.ti.com

## NAME

constant − Perl pragma to declare constants

## SYNOPSIS

```
use constant BUFFER_SIZE    => 4096;
use constant ONE_YEAR       => 365.2425 * 24 * 60 * 60;
use constant PI             => 4 * atan2 1, 1;
use constant DEBUGGING      => 0;
use constant ORACLE         => 'oracle@cs.indiana.edu';
use constant USERNAME       => scalar getpwuid($<);
use constant USERINFO       => getpwuid($<);

sub deg2rad { PI * $_[0] / 180 }

print "This line does nothing"              unless DEBUGGING;
```

## DESCRIPTION

This will declare a symbol to be a constant with the given scalar or list value.

When you declare a constant such as `PI` using the method shown above, each machine your script runs upon can have as many digits of accuracy as it can use. Also, your program will be easier to read, more likely to be maintained (and maintained correctly), and far less likely to send a space probe to the wrong planet because nobody noticed the one equation in which you wrote `3.14195`.

## NOTES

The value or values are evaluated in a list context. You may override this with `scalar` as shown above.

These constants do not directly interpolate into double-quotish strings, although you may do so indirectly. (See the *perlref* manpage for details about how this works.)

```
print "The value of PI is @{[ PI ]}.\n";
```

List constants are returned as lists, not as arrays.

```
$homedir = USERINFO[7];             # WRONG
$homedir = (USERINFO)[7];           # Right
```

The use of all caps for constant names is merely a convention, although it is recommended in order to make constants stand out and to help avoid collisions with other barewords, keywords, and subroutine names. Constant names must begin with a letter.

Constant symbols are package scoped (rather than block scoped, as `use strict` is). That is, you can refer to a constant from package Other as `Other::CONST`.

As with all `use` directives, defining a constant happens at compile time. Thus, it's probably not correct to put a constant declaration inside of a conditional statement (like `if ($foo) { use constant ... }`).

Omitting the value for a symbol gives it the value of `undef` in a scalar context or the empty list, `()`, in a list context. This isn't so nice as it may sound, though, because in this case you must either quote the symbol name, or use a big arrow, (=>), with nothing to point to. It is probably best to declare these explicitly.

```
use constant UNICORNS       => ();
use constant LOGFILE        => undef;
```

The result from evaluating a list constant in a scalar context is not documented, and is **not** guaranteed to be any particular value in the future. In particular, you should not rely upon it being the number of elements in the list, especially since it is not **necessarily** that value in the current implementation.

Magical values, tied values, and references can be made into constants at compile time, allowing for way cool stuff like this. (These error numbers aren't totally portable, alas.)

```
use constant E2BIG => ($! = 7);
print   E2BIG, "\n";        # something like "Arg list too long"
print 0+E2BIG, "\n";        # "7"
```

## TECHNICAL NOTE

In the current implementation, scalar constants are actually inlinable subroutines. As of version 5.004 of Perl, the appropriate scalar constant is inserted directly in place of some subroutine calls, thereby saving the overhead of a subroutine call. See the section on *Constant Functions* in the *perlsub* manpage for details about how and when this happens.

## BUGS

In the current version of Perl, list constants are not inlined and some symbols may be redefined without generating a warning.

It is not possible to have a subroutine or keyword with the same name as a constant. This is probably a Good Thing.

Unlike constants in some languages, these cannot be overridden on the command line or via environment variables.

You can get into trouble if you use constants in a context which automatically quotes barewords (as is true for any subroutine call). For example, you can't say `$hash{CONSTANT}` because `CONSTANT` will be interpreted as a string. Use `$hash{CONSTANT()}` or `$hash{+CONSTANT}` to prevent the bareword quoting mechanism from kicking in. Similarly, since the => operator quotes a bareword immediately to its left you have to say `CONSTANT() => 'value'` instead of `CONSTANT => 'value'`.

## AUTHOR

Tom Phoenix, <*rootbeer@teleport.com*>, with help from many other folks.

## COPYRIGHT

## NAME

diagnostics − Perl compiler pragma to force verbose warning diagnostics

splain − standalone program to do the same thing

## SYNOPSIS

As a pragma:

```
use diagnostics;
use diagnostics -verbose;

enable  diagnostics;
disable diagnostics;
```

Aa a program:

```
perl program 2>diag.out
splain [-v] [-p] diag.out
```

## DESCRIPTION

### The `diagnostics` Pragma

This module extends the terse diagnostics normally emitted by both the perl compiler and the perl interpeter, augmenting them with the more explicative and endearing descriptions found in the *perldiag* manpage. Like the other pragmata, it affects the compilation phase of your program rather than merely the execution phase.

To use in your program as a pragma, merely invoke

```
use diagnostics;
```

at the start (or near the start) of your program. (Note that this *does* enable perl's **−w** flag.) Your whole compilation will then be *subject* (ed :−) to the enhanced diagnostics. These still go out **STDERR**.

Due to the interaction between runtime and compiletime issues, and because it's probably not a very good idea anyway, you may not use `no diagnostics` to turn them off at compiletime. However, you may control there behaviour at runtime using the *disable()* and *enable()* methods to turn them off and on respectively.

The **−verbose** flag first prints out the the *perldiag* manpage introduction before any other diagnostics. The `$diagnostics::PRETTY` variable can generate nicer escape sequences for pagers.

### The *splain* Program

While apparently a whole nuther program, *splain* is actually nothing more than a link to the (executable) *diagnostics.pm* module, as well as a link to the *diagnostics.pod* documentation. The **−v** flag is like the `use diagnostics -verbose` directive. The **−p** flag is like the `$diagnostics::PRETTY` variable. Since you're post-processing with *splain*, there's no sense in being able to *enable()* or *disable()* processing.

Output from *splain* is directed to **STDOUT**, unlike the pragma.

## EXAMPLES

The following file is certain to trigger a few errors at both runtime and compiletime:

```
use diagnostics;
print NOWHERE "nothing\n";
print STDERR "\n\tThis message should be unadorned.\n";
warn "\tThis is a user warning";
print "\nDIAGNOSTIC TESTER: Please enter a <CR> here: ";
my $a, $b = scalar <STDIN>;
print "\n";
print $x/$y;
```

If you prefer to run your program first and look at its problem afterwards, do this:

```
perl -w test.pl 2>test.out
./splain < test.out
```

Note that this is not in general possible in shells of more dubious heritage, as the theoretical

```
(perl -w test.pl >/dev/tty) >& test.out
./splain < test.out
```

Because you just moved the existing **stdout** to somewhere else.

If you don't want to modify your source code, but still have on-the-fly warnings, do this:

```
exec 3>&1; perl -w test.pl 2>&1 1>&3 3>&- | splain 1>&2 3>&-
```

Nifty, eh?

If you want to control warnings on the fly, do something like this. Make sure you do the `use` first, or you won't be able to get at the *enable()* or *disable()* methods.

```
use diagnostics; # checks entire compilation phase
    print "\ntime for 1st bogus diags: SQUAWKINGS\n";
    print BOGUS1 'nada';
    print "done with 1st bogus\n";

disable diagnostics; # only turns off runtime warnings
    print "\ntime for 2nd bogus: (squelched)\n";
    print BOGUS2 'nada';
    print "done with 2nd bogus\n";

enable diagnostics; # turns back on runtime warnings
    print "\ntime for 3rd bogus: SQUAWKINGS\n";
    print BOGUS3 'nada';
    print "done with 3rd bogus\n";

disable diagnostics;
    print "\ntime for 4th bogus: (squelched)\n";
    print BOGUS4 'nada';
    print "done with 4th bogus\n";
```

## INTERNALS

Diagnostic messages derive from the *perldiag.pod* file when available at runtime. Otherwise, they may be embedded in the file itself when the splain package is built. See the *Makefile* for details.

If an extant `$SIG{__WARN__}` handler is discovered, it will continue to be honored, but only after the *diagnostics::splainthis()* function (the module's `$SIG{__WARN__}` interceptor) has had its way with your warnings.

There is a `$diagnostics::DEBUG` variable you may set if you're desperately curious what sorts of things are being intercepted.

```
BEGIN { $diagnostics::DEBUG = 1 }
```

## BUGS

Not being able to say "no diagnostics" is annoying, but may not be insurmountable.

The `-pretty` directive is called too late to affect matters. You have to do this instead, and *before* you load the module.

```
BEGIN { $diagnostics::PRETTY = 1 }
```

I could start up faster by delaying compilation until it should be needed, but this gets a "panic: top_level" when using the pragma form in Perl 5.001e.

While it's true that this documentation is somewhat subserious, if you use a program named *splain*, you should expect a bit of whimsy.

## AUTHOR

Tom Christiansen <*tchrist@mox.perl.com*>, 25 June 1995.

**NAME**

fields − compile-time class fields

**SYNOPSIS**

```
{
    package Foo;
    use fields qw(foo bar _private);
}
...
my Foo $var = new Foo;
$var->{foo} = 42;

# This will generate a compile-time error.
$var->{zap} = 42;

{
    package Bar;
    use base 'Foo';
    use fields 'bar';             # hides Foo->{bar}
    use fields qw(baz _private);  # not shared with Foo
}
```

**DESCRIPTION**

The `fields` pragma enables compile-time verified class fields. It does so by updating the `%FIELDS` hash in the calling package.

If a typed lexical variable holding a reference is used to access a hash element and the `%FIELDS` hash of the given type exists, then the operation is turned into an array access at compile time. The `%FIELDS` hash map from hash element names to the array indices. If the hash element is not present in the `%FIELDS` hash, then a compile-time error is signaled.

Since the `%FIELDS` hash is used at compile-time, it must be set up at compile-time too. This is made easier with the help of the 'fields' and the 'base' pragma modules. The 'base' pragma will copy fields from base classes and the 'fields' pragma adds new fields. Field names that start with an underscore character are made private to a class and are not visible to subclasses. Inherited fields can be overridden but will generate a warning if used together with the `−w` switch.

The effect of all this is that you can have objects with named fields which are as compact and as fast arrays to access. This only works as long as the objects are accessed through properly typed variables. For untyped access to work you have to make sure that a reference to the proper `%FIELDS` hash is assigned to the 0'th element of the array object (so that the objects can be treated like an pseudo-hash). A constructor like this does the job:

```
sub new
{
    my $class = shift;
    no strict 'refs';
    my $self = bless [\%{"$class\::FIELDS"], $class;
    $self;
}
```

**SEE ALSO**

the *base* manpage, the section on *Pseudo-hashes: Using an array as a hash* in the *perlref* manpage

## NAME

integer − Perl pragma to compute arithmetic in integer instead of double

## SYNOPSIS

```
use integer;
$x = 10/3;
# $x is now 3, not 3.33333333333333333
```

## DESCRIPTION

This tells the compiler to use integer operations from here to the end of the enclosing BLOCK. On many machines, this doesn't matter a great deal for most computations, but on those without floating point hardware, it can make a big difference.

Note that this affects the operations, not the numbers. If you run this code

```
use integer;
$x = 1.5;
$y = $x + 1;
$z = -1.5;
```

you'll be left with `$x == 1.5`, `$y == 2` and `$z == -1`. The `$z` case happens because unary − counts as an operation.

See the section on *Pragmatic Modules* in the *perlmod* manpage.

**NAME**

less – perl pragma to request less of something from the compiler

**SYNOPSIS**

```
use less;  # unimplemented
```

**DESCRIPTION**

Currently unimplemented, this may someday be a compiler directive to make certain trade-offs, such as perhaps

```
use less 'memory';
use less 'CPU';
use less 'fat';
```

## NAME

lib − manipulate `@INC` at compile time

## SYNOPSIS

```
use lib LIST;

no lib LIST;
```

## DESCRIPTION

This is a small simple module which simplifies the manipulation of `@INC` at compile time.

It is typically used to add extra directories to perl's search path so that later `use` or `require` statements will find modules which are not located on perl's default search path.

### ADDING DIRECTORIES TO @INC

The parameters to `use lib` are added to the start of the perl search path. Saying

```
use lib LIST;
```

is *almost* the same as saying

```
BEGIN { unshift(@INC, LIST) }
```

For each directory in LIST (called `$dir` here) the lib module also checks to see if a directory called `$dir`/$archname/auto exists. If so the `$dir`/$archname directory is assumed to be a corresponding architecture specific directory and is added to `@INC` in front of `$dir`.

If LIST includes both `$dir` and `$dir`/$archname then `$dir`/$archname will be added to `@INC` twice (if `$dir`/$archname/auto exists).

### DELETING DIRECTORIES FROM @INC

You should normally only add directories to `@INC`. If you need to delete directories from `@INC` take care to only delete those which you added yourself or which you are certain are not needed by other modules in your script. Other modules may have added directories which they need for correct operation.

By default the `no lib` statement deletes the *first* instance of each named directory from `@INC`. To delete multiple instances of the same name from `@INC` you can specify the name multiple times.

To delete *all* instances of *all* the specified names from `@INC` you can specify ':ALL' as the first parameter of `no lib`. For example:

```
no lib qw(:ALL .);
```

For each directory in LIST (called `$dir` here) the lib module also checks to see if a directory called `$dir`/$archname/auto exists. If so the `$dir`/$archname directory is assumed to be a corresponding architecture specific directory and is also deleted from `@INC`.

If LIST includes both `$dir` and `$dir`/$archname then `$dir`/$archname will be deleted from `@INC` twice (if `$dir`/$archname/auto exists).

### RESTORING ORIGINAL @INC

When the lib module is first loaded it records the current value of `@INC` in an array `@lib::ORIG_INC`. To restore `@INC` to that value you can say

```
@INC = @lib::ORIG_INC;
```

## SEE ALSO

FindBin − optional module which deals with paths relative to the source file.

**AUTHOR**

Tim Bunce, 2nd June 1995.

## NAME

locale − Perl pragma to use and avoid POSIX locales for built-in operations

## SYNOPSIS

```
@x = sort @y;        # ASCII sorting order
{
    use locale;
    @x = sort @y;    # Locale-defined sorting order
}
@x = sort @y;        # ASCII sorting order again
```

## DESCRIPTION

This pragma tells the compiler to enable (or disable) the use of POSIX locales for built-in operations (LC_CTYPE for regular expressions, and LC_COLLATE for string comparison). Each ‘‘use locale’’ or ‘‘no locale’’ affects statements to the end of the enclosing BLOCK.

## NAME

ops − Perl pragma to restrict unsafe operations when compiling

## SYNOPSIS

```
perl -Mops=:default ...    # only allow reasonably safe operations

perl -M-ops=system ...     # disable the 'system' opcode
```

## DESCRIPTION

Since the ops pragma currently has an irreversible global effect, it is only of significant practical use with the `-M` option on the command line.

See the the *Opcode* manpage module for information about opcodes, optags, opmasks and important information about safety.

## SEE ALSO

*Opcode* (3), *Safe* (3), *perlrun* (3)

## NAME

overload − Package for overloading perl operations

## SYNOPSIS

```
package SomeThing;

use overload
    '+' => \&myadd,
    '-' => \&mysub;
    # etc
...

package main;
$a = new SomeThing 57;
$b=5+$a;
...
if (overload::Overloaded $b) {...}
...
$strval = overload::StrVal $b;
```

## CAVEAT SCRIPTOR

Overloading of operators is a subject not to be taken lightly. Neither its precise implementation, syntax, nor semantics are 100% endorsed by Larry Wall. So any of these may be changed at some point in the future.

## DESCRIPTION

### Declaration of overloaded functions

The compilation directive

```
package Number;
use overload
    "+" => \&add,
    "*=" => "muas";
```

declares function *Number::add()* for addition, and method *muas()* in the "class" `Number` (or one of its base classes) for the assignment form `*=` of multiplication.

Arguments of this directive come in (key, value) pairs. Legal values are values legal inside a `&{ ... }` call, so the name of a subroutine, a reference to a subroutine, or an anonymous subroutine will all work. Note that values specified as strings are interpreted as methods, not subroutines. Legal keys are listed below.

The subroutine `add` will be called to execute `$a+$b` if `$a` is a reference to an object blessed into the package `Number`, or if `$a` is not an object from a package with defined mathemagic addition, but `$b` is a reference to a `Number`. It can also be called in other situations, like `$a+=7`, or `$a++`. See the section on *MAGIC AUTOGENERATION*. (Mathemagical methods refer to methods triggered by an overloaded mathematical operator.)

Since overloading respects inheritance via the `@ISA` hierarchy, the above declaration would also trigger overloading of + and `*=` in all the packages which inherit from `Number`.

### Calling Conventions for Binary Operations

The functions specified in the `use overload ...` directive are called with three (in one particular case with four, see the section on *Last Resort*) arguments. If the corresponding operation is binary, then the first two arguments are the two arguments of the operation. However, due to general object calling conventions, the first argument should always be an object in the package, so in the situation of `7+$a`, the order of the arguments is interchanged. It probably does not matter when implementing the addition method, but whether the arguments are reversed is vital to the subtraction method. The method can query this information by examining the third argument, which can take three different values:

FALSE the order of arguments is as in the current operation.

TRUE the arguments are reversed.

`undef` the current operation is an assignment variant (as in `$a+=7`), but the usual function is called instead. This additional information can be used to generate some optimizations. Compare the section on *Calling Conventions for Mutators*.

**Calling Conventions for Unary Operations**

Unary operation are considered binary operations with the second argument being `undef`. Thus the functions that overloads `{"++"}` is called with arguments `($a,undef,'')` when `$a++` is executed.

**Calling Conventions for Mutators**

Two types of mutators have different calling conventions:

`++` and `--`

The routines which implement these operators are expected to actually *mutate* their arguments. So, assuming that `$obj` is a reference to a number,

```
sub incr { my $n = $ {$_[0]}; ++$n; $_[0] = bless \$n}
```

is an appropriate implementation of overloaded `++`. Note that

```
sub incr { ++$ {$_[0]} ; shift }
```

is OK if used with preincrement and with postincrement. (In the case of postincrement a copying will be performed, see the section on *Copy Constructor*.)

`x=` and other assignment versions

There is nothing special about these methods. They may change the value of their arguments, and may leave it as is. The result is going to be assigned to the value in the left-hand-side if different from this value.

This allows for the same method to be used as averloaded `+=` and `+`. Note that this is *allowed*, but not recommended, since by the semantic of the section on *Fallback* Perl will call the method for `+` anyway, if `+=` is not overloaded.

**Warning.** Due to the presense of assignment versions of operations, routines which may be called in assignment context may create self-referencial structures. Currently Perl will not free self-referential structures until cycles are `explicitly` broken. You may get problems when traversing your structures too.

Say,

```
use overload '+' => sub { bless [ \$_[0], \$_[1] ] };
```

is asking for trouble, since for code `$obj += $foo` the subroutine is called as `$obj = add($obj, $foo, undef)`, or `$obj = [\$obj, \$foo]`. If using such a subroutine is an important optimization, one can overload `+=` explicitly by a non-"optimized'' version, or switch to non-optimized version if `not defined $_[2]` (see the section on *Calling Conventions for Binary Operations*).

Even if no *explicit* assignment-variants of operators are present in the script, they may be generated by the optimizer. Say, `",$obj,"` or `',' . $obj . ','` may be both optimized to

```
my $tmp = ',' . $obj;    $tmp .= ',';
```

### Overloadable Operations

The following symbols can be specified in `use overload` directive:

- *Arithmetic operations*

```
"+", "+=", "-", "-=", "*", "*=", "/", "/=", "%", "%=",
"**", "**=", "<<", "<<=", ">>", ">>=", "x", "x=", ".", ".=",
```

For these operations a substituted non-assignment variant can be called if the assignment variant is not available. Methods for operations "+", "−", "+=", and "−=" can be called to automatically generate increment and decrement methods. The operation "−" can be used to autogenerate missing methods for unary minus or `abs`.

See the section on *MAGIC AUTOGENERATION*, the section on *Calling Conventions for Mutators* and the section on *Calling Conventions for Binary Operations*) for details of these substitutions.

- *Comparison operations*

```
"<",  "<=", ">",  ">=", "==", "!=", "<=>",
"lt", "le", "gt", "ge", "eq", "ne", "cmp",
```

If the corresponding "spaceship" variant is available, it can be used to substitute for the missing operation. During `sort`ing arrays, `cmp` is used to compare values subject to `use overload`.

- *Bit operations*

```
"&", "^", "|", "neg", "!", "~",
```

"`neg`" stands for unary minus. If the method for `neg` is not specified, it can be autogenerated using the method for subtraction. If the method for "`!`" is not specified, it can be autogenerated using the methods for "`bool`", or "`\"\"`", or "`0+`".

- *Increment and decrement*

```
"++", "--",
```

If undefined, addition and subtraction methods can be used instead. These operations are called both in prefix and postfix form.

- *Transcendental functions*

```
"atan2", "cos", "sin", "exp", "abs", "log", "sqrt",
```

If `abs` is unavailable, it can be autogenerated using methods for "<" or "<=>" combined with either unary minus or subtraction.

- *Boolean, string and numeric conversion*

```
"bool", "\"\"", "0+",
```

If one or two of these operations are unavailable, the remaining ones can be used instead. `bool` is used in the flow control operators (like `while`) and for the ternary "`?:`" operation. These functions can return any arbitrary Perl value. If the corresponding operation for this value is overloaded too, that operation will be called again with this value.

- *Special*

```
"nomethod", "fallback", "=",
```

see the section on *SPECIAL SYMBOLS FOR* `use overload`.

See the section on *Fallback* for an explanation of when a missing method can be autogenerated.

A computer-readable form of the above table is available in the hash `%overload::ops`, with values being space-separated lists of names:

```
with_assign      => '+ - * / % ** << >> x .',
assign           => '+= -= *= /= %= **= <<= >>= x= .=',
str_comparison   => '< <= > >= == !=',
'3way_comparison'=> '<=> cmp',
num_comparison   => 'lt le gt ge eq ne',
binary           => '& | ^',
unary            => 'neg ! ~',
mutators         => '++ --',
func             => 'atan2 cos sin exp abs log sqrt',
conversion       => 'bool "" 0+',
special          => 'nomethod fallback ='
```

### Inheritance and overloading

Inheritance interacts with overloading in two ways.

Strings as values of `use overload` directive

If `value` in

```
use overload key => value;
```

is a string, it is interpreted as a method name.

Overloading of an operation is inherited by derived classes

Any class derived from an overloaded class is also overloaded. The set of overloaded methods is the union of overloaded methods of all the ancestors. If some method is overloaded in several ancestor, then which description will be used is decided by the usual inheritance rules:

If `A` inherits from `B` and `C` (in this order), `B` overloads `+` with `\&D::plus_sub`, and `C` overloads `+` by `"plus_meth"`, then the subroutine `D::plus_sub` will be called to implement operation `+` for an object in package `A`.

Note that since the value of the `fallback` key is not a subroutine, its inheritance is not governed by the above rules. In the current implementation, the value of `fallback` in the first overloaded ancestor is used, but this is accidental and subject to change.

## SPECIAL SYMBOLS FOR `use overload`

Three keys are recognized by Perl that are not covered by the above description.

### Last Resort

`"nomethod"` should be followed by a reference to a function of four parameters. If defined, it is called when the overloading mechanism cannot find a method for some operation. The first three arguments of this function coincide with the arguments for the corresponding method if it were found, the fourth argument is the symbol corresponding to the missing method. If several methods are tried, the last one is used. Say, `1-$a` can be equivalent to

```
&nomethodMethod($a,1,1,"-")
```

if the pair `"nomethod" => "nomethodMethod"` was specified in the `use overload` directive.

If some operation cannot be resolved, and there is no function assigned to `"nomethod"`, then an exception will be raised via *die()*-- unless `"fallback"` was specified as a key in `use overload` directive.

### Fallback

The key `"fallback"` governs what to do if a method for a particular operation is not found. Three different cases are possible depending on the value of `"fallback"`:

- `undef` Perl tries to use a substituted method (see the section on *MAGIC AUTOGENERATION*). If this fails, it then tries to calls `"nomethod"` value; if missing, an exception will be raised.
- TRUE The same as for the `undef` value, but no exception is raised. Instead, it silently reverts to what it would have done were there no `use overload` present.
- defined, but FALSE
  No autogeneration is tried. Perl tries to call `"nomethod"` value, and if this is missing, raises an exception.

**Note.** `"fallback"` inheritance via `@ISA` is not carved in stone yet, see the section on *Inheritance and overloading*.

### Copy Constructor

The value for `"="` is a reference to a function with three arguments, i.e., it looks like the other values in `use overload`. However, it does not overload the Perl assignment operator. This would go against Camel hair.

This operation is called in the situations when a mutator is applied to a reference that shares its object with some other reference, such as

```
$a=$b;
++$a;
```

To make this change `$a` and not change `$b`, a copy of `$$a` is made, and `$a` is assigned a reference to this new object. This operation is done during execution of the `++$a`, and not during the assignment, (so before the increment `$$a` coincides with `$$b`). This is only done if `++` is expressed via a method for `'++'` or `'+='` (or `nomethod`). Note that if this operation is expressed via `'+'` a nonmutator, i.e., as in

```
$a=$b;
$a=$a+1;
```

then `$a` does not reference a new copy of `$$a`, since $$a does not appear as lvalue when the above code is executed.

If the copy constructor is required during the execution of some mutator, but a method for `'='` was not specified, it can be autogenerated as a string copy if the object is a plain scalar.

**Example**

The actually executed code for

```
$a=$b;
Something else which does not modify $a or $b....
++$a;
```

may be

```
$a=$b;
Something else which does not modify $a or $b....
$a = $a->clone(undef,"");
$a->incr(undef,"");
```

if `$b` was mathemagical, and `'++'` was overloaded with `\&incr`, `'='` was overloaded with `\&clone`.

Same behaviour is triggered by `$b = $a++`, which is consider a synonim for `$b = $a; ++$a`.

**MAGIC AUTOGENERATION**

If a method for an operation is not found, and the value for `"fallback"` is TRUE or undefined, Perl tries to autogenerate a substitute method for the missing operation based on the defined operations. Autogenerated method substitutions are possible for the following operations:

*Assignment forms of arithmetic operations*
: `$a+=$b` can use the method for `"+"` if the method for `"+="` is not defined.

*Conversion operations*
: String, numeric, and boolean conversion are calculated in terms of one another if not all of them are defined.

*Increment and decrement*
: The `++$a` operation can be expressed in terms of `$a+=1` or `$a+1`, and `$a--` in terms of `$a-=1` and `$a-1`.

`abs($a)`
: can be expressed in terms of `$a<0` and `-$a` (or `0-$a`).

*Unary minus*
: can be expressed in terms of subtraction.

*Negation*
: `!` and `not` can be expressed in terms of boolean conversion, or string or numerical conversion.

*Concatenation*
: can be expressed in terms of string conversion.

*Comparison operations*
: can be expressed in terms of its ‘‘spaceship’’ counterpart: either <=> or `cmp`:

```
<, >, <=, >=, ==, !=        in terms of <=>
lt, gt, le, ge, eq, ne      in terms of cmp
```

*Copy operator*
: can be expressed in terms of an assignment to the dereferenced value, if this value is a scalar and not a reference.

**Losing overloading**

The restriction for the comparison operation is that even if, for example, ‘`cmp`’ should return a blessed reference, the autogenerated ‘`lt`’ function will produce only a standard logical value based on the numerical value of the result of ‘`cmp`’. In particular, a working numeric conversion is needed in this case (possibly expressed in terms of other conversions).

Similarly, `.=` and `x=` operators lose their mathemagical properties if the string conversion substitution is applied.

When you *chop()* a mathemagical object it is promoted to a string and its mathemagical properties are lost. The same can happen with other operations as well.

**Run-time Overloading**

Since all `use` directives are executed at compile-time, the only way to change overloading during run-time is to

```
eval 'use overload "+" => \&addmethod';
```

You can also use

```
eval 'no overload "+", "--", "<="';
```

though the use of these constructs during run-time is questionable.

**Public functions**

Package `overload.pm` provides the following public functions:

overload::StrVal(arg)
: Gives string value of `arg` as in absence of stringify overloading.

overload::Overloaded(arg)
: Returns true if `arg` is subject to overloading of some operations.

overload::Method(obj,op)
: Returns `undef` or a reference to the method that implements `op`.

**Overloading constants**

For some application Perl parser mangles constants too much. It is possible to hook into this process via *overload::constant()* and *overload::remove_constant()* functions.

These functions take a hash as an argument. The recognized keys of this hash are

integer to overload integer constants,

float to overload floating point constants,

binary to overload octal and hexadecimal constants,

q to overload `q`−quoted strings, constant pieces of `qq`− and `qx`−quoted strings and here-documents,

qr to overload constant pieces of regular expressions.

The corresponding values are references to functions which take three arguments: the first one is the *initial* string form of the constant, the second one is how Perl interprets this constant, the third one is how the constant is used. Note that the initial string form does not contain string delimiters, and has backslashes in backslash-delimiter combinations stripped (thus the value of delimiter is not relevant for processing of this string). The return value of this function is how this constant is going to be interpreted by Perl. The third argument is undefined unless for overloaded `q`− and `qr`− constants, it is `q` in single-quote context (comes from strings, regular expressions, and single-quote HERE documents), it is `tr` for arguments of `tr`/`y` operators, it is `s` for right-hand side of `s`−operator, and it is `qq` otherwise.

Since an expression `"ab$cd,,"` is just a shortcut for `'ab' . $cd . ',,'`, it is expected that overloaded constant strings are equipped with reasonable overloaded catenation operator, otherwise absurd results will result. Similarly, negative numbers are considered as negations of positive constants.

Note that it is probably meaningless to call the functions *overload::constant()* and *overload::remove_constant()* from anywhere but *import()* and *unimport()* methods. From these methods they may be called as

```
sub import {
  shift;
  return unless @_;
  die "unknown import: @_" unless @_ == 1 and $_[0] eq ':constant';
  overload::constant integer => sub {Math::BigInt->new(shift)};
}
```

**BUGS** Currently overloaded-ness of constants does not propagate into `eval '...'`.

**IMPLEMENTATION**

What follows is subject to change RSN.

The table of methods for all operations is cached in magic for the symbol table hash for the package. The cache is invalidated during processing of `use overload`, `no overload`, new function definitions, and changes in `@ISA`. However, this invalidation remains unprocessed until the next `bless`ing into the package. Hence if you want to change overloading structure dynamically, you'll need an additional (fake) `bless`ing to update the table.

(Every SVish thing has a magic queue, and magic is an entry in that queue. This is how a single variable may participate in multiple forms of magic simultaneously. For instance, environment variables regularly have two forms at once: their `%ENV` magic and their taint magic. However, the magic which implements overloading is applied to the stashes, which are rarely used directly, thus should not slow down Perl.)

If an object belongs to a package using overload, it carries a special flag. Thus the only speed penalty during arithmetic operations without overloading is the checking of this flag.

In fact, if `use overload` is not present, there is almost no overhead for overloadable operations, so most programs should not suffer measurable performance penalties. A considerable effort was made to minimize the overhead when overload is used in some package, but the arguments in question do not belong to packages using overload. When in doubt, test your speed with `use overload` and without it. So far there have been no reports of substantial speed degradation if Perl is compiled with optimization turned on.

There is no size penalty for data if overload is not used. The only size penalty if overload is used in some package is that *all* the packages acquire a magic during the next `bless`ing into the package. This magic is three-words-long for packages without overloading, and carries the cache tabel if the package is overloaded.

Copying (`$a=$b`) is shallow; however, a one-level-deep copying is carried out before any operation that can imply an assignment to the object `$a` (or `$b`) refers to, like `$a++`. You can override this behavior by defining your own copy constructor (see the section on *Copy Constructor*).

It is expected that arguments to methods that are not explicitly supposed to be changed are constant (but this is not enforced).

**Metaphor clash**

One may wonder why the semantic of overloaded = is so counterintuive. If it *looks* counterintuive to you, you are subject to a metaphor clash.

Here is a Perl object metaphor:

*object is a reference to blessed data*

and an arithmetic metaphor:

*object is a thing by itself*.

The *main* problem of overloading = is the fact that these metaphors imply different actions on the assignment `$a = $b` if `$a` and `$b` are objects. Perl-think implies that `$a` becomes a reference to whatever `$b` was referencing. Arithmetic-think implies that the value of "object" `$a` is changed to become the value of the object `$b`, preserving the fact that `$a` and `$b` are separate entities.

The difference is not relevant in the absence of mutators. After a Perl-way assignment an operation which mutates the data referenced by `$a` would change the data referenced by `$b` too. Effectively, after `$a = $b` values of `$a` and `$b` become *indistinguishable*.

On the other hand, anyone who has used algebraic notation knows the expressive power of the arithmetic metaphor. Overloading works hard to enable this metaphor while preserving the Perlian way as far as possible. Since it is not not possible to freely mix two contradicting metaphors, overloading allows the arithmetic way to write things *as far as all the mutators are called via overloaded access only*. The way it is done is described in the section on *Copy Constructor*.

If some mutator methods are directly applied to the overloaded values, one may need to *explicitly unlink* other values which references the same value:

```
$a = new Data 23;
...
$b = $a;            # $b is "linked" to $a
...
$a = $a->clone;     # Unlink $b from $a
$a->increment_by(4);
```

Note that overloaded access makes this transparent:

```
$a = new Data 23;
$b = $a;            # $b is "linked" to $a
$a += 4;            # would unlink $b automagically
```

However, it would not make

```
$a = new Data 23;
$a = 4;              # Now $a is a plain 4, not 'Data'
```

preserve "objectness" of `$a`. But Perl *has* a way to make assignments to an object do whatever you want. It is just not the overload, but *tie()*ing interface (see the `tie` entry in the *perlfunc* manpage). Adding a *FETCH()* method which returns the object itself, and *STORE()* method which changes the value of the object, one can reproduce the arithmetic metaphor in its completeness, at least for variables which were *tie()*d from the start.

(Note that a workaround for a bug may be needed, see the section on *BUGS*.)

**Cookbook**

Please add examples to what follows!

### Two-face scalars

Put this in *two_face.pm* in your Perl library directory:

```
package two_face;              # Scalars with separate string and
                               # numeric values.
sub new { my $p = shift; bless [@_], $p }
use overload '""' => \&str, '0+' => \&num, fallback => 1;
sub num {shift->[1]}
sub str {shift->[0]}
```

Use it as follows:

```
require two_face;
my $seven = new two_face ("vii", 7);
printf "seven=$seven, seven=%d, eight=%d\n", $seven, $seven+1;
print "seven contains `i'\n" if $seven =~ /i/;
```

(The second line creates a scalar which has both a string value, and a numeric value.) This prints:

```
seven=vii, seven=7, eight=8
seven contains `i'
```

### Symbolic calculator

Put this in *symbolic.pm* in your Perl library directory:

```
package symbolic;              # Primitive symbolic calculator
use overload nomethod => \&wrap;

sub new { shift; bless ['n', @_] }
sub wrap {
  my ($obj, $other, $inv, $meth) = @_;
  ($obj, $other) = ($other, $obj) if $inv;
  bless [$meth, $obj, $other];
}
```

This module is very unusual as overloaded modules go: it does not provide any usual overloaded operators, instead it provides the the section on *Last Resort* operator `nomethod`. In this example the corresponding subroutine returns an object which encupsulates operations done over the objects: `new symbolic 3` contains `['n', 3]`, `2 + new symbolic 3` contains `['+', 2, ['n', 3]]`.

Here is an example of the script which "calculates" the side of circumscribed octagon using the above package:

```
require symbolic;
my $iter = 1;                   # 2**($iter+2) = 8
my $side = new symbolic 1;
my $cnt = $iter;

while ($cnt--) {
  $side = (sqrt(1 + $side**2) - 1)/$side;
}
print "OK\n";
```

The value of `$side` is

```
['/', ['-', ['sqrt', ['+', 1, ['**', ['n', 1], 2]],
                     undef], 1], ['n', 1]]
```

Note that while we obtained this value using a nice little script, there is no simple way to *use* this value. In fact this value may be inspected in debugger (see the *perldebug* manpage), but ony if `bareStringify` **O**ption is set, and not via `p` command.

If one attempts to print this value, then the overloaded operator `""` will be called, which will call `nomethod` operator. The result of this operator will be stringified again, but this result is again of type `symbolic`, which will lead to an infinite loop.

Add a pretty-printer method to the module *symbolic.pm*:

```
sub pretty {
  my ($meth, $a, $b) = @{+shift};
  $a = 'u' unless defined $a;
  $b = 'u' unless defined $b;
  $a = $a->pretty if ref $a;
  $b = $b->pretty if ref $b;
  "[$meth $a $b]";
}
```

Now one can finish the script by

```
print "side = ", $side->pretty, "\n";
```

The method `pretty` is doing object-to-string conversion, so it is natural to overload the operator `""` using this method. However, inside such a method it is not necessary to pretty-print the *components* `$a` and `$b` of an object. In the above subroutine `"[$meth $a $b]"` is a catenation of some strings and components `$a` and `$b`. If these components use overloading, the catenation operator will look for an overloaded operator `.`, if not present, it will look for an overloaded operator `""`. Thus it is enough to use

```
use overload nomethod => \&wrap, '""' => \&str;
sub str {
  my ($meth, $a, $b) = @{+shift};
  $a = 'u' unless defined $a;
  $b = 'u' unless defined $b;
  "[$meth $a $b]";
}
```

Now one can change the last line of the script to

```
print "side = $side\n";
```

which outputs

```
side = [/ [- [sqrt [+ 1 [** [n 1 u] 2]] u] 1] [n 1 u]]
```

and one can inspect the value in debugger using all the possible methods.

Something is is still amiss: consider the loop variable `$cnt` of the script. It was a number, not an object. We cannot make this value of type `symbolic`, since then the loop will not terminate.

Indeed, to terminate the cycle, the `$cnt` should become false. However, the operator `bool` for checking falsity is overloaded (this time via overloaded `""`), and returns a long string, thus any object of type `symbolic` is true. To overcome this, we need a way to compare an object to 0. In fact, it is easier to write a numeric conversion routine.

Here is the text of *symbolic.pm* with such a routine added (and slightly modifed *str()*):

```
package symbolic;              # Primitive symbolic calculator
use overload
  nomethod => \&wrap, '""' => \&str, '0+' => \&num;

sub new { shift; bless ['n', @_] }
sub wrap {
  my ($obj, $other, $inv, $meth) = @_;
  ($obj, $other) = ($other, $obj) if $inv;
  bless [$meth, $obj, $other];
}
sub str {
  my ($meth, $a, $b) = @{+shift};
  $a = 'u' unless defined $a;
  if (defined $b) {
    "[$meth $a $b]";
  } else {
    "[$meth $a]";
  }
}
my %subr = ( n => sub {$_[0]},
             sqrt => sub {sqrt $_[0]},
             '-' => sub {shift() - shift()},
             '+' => sub {shift() + shift()},
             '/' => sub {shift() / shift()},
             '*' => sub {shift() * shift()},
             '**' => sub {shift() ** shift()},
           );
sub num {
  my ($meth, $a, $b) = @{+shift};
  my $subr = $subr{$meth}
    or die "Do not know how to ($meth) in symbolic";
  $a = $a->num if ref $a eq __PACKAGE__;
  $b = $b->num if ref $b eq __PACKAGE__;
  $subr->($a,$b);
}
```

All the work of numeric conversion is done in `%subr` and *num()*. Of course, `%subr` is not complete, it contains only operators used in teh example below. Here is the extra-credit question: why do we need an explicit recursion in *num()*? (Answer is at the end of this section.)

Use this module like this:

```
require symbolic;
my $iter = new symbolic 2;    # 16-gon
my $side = new symbolic 1;
my $cnt = $iter;

while ($cnt) {
  $cnt = $cnt - 1;            # Mutator `--' not implemented
  $side = (sqrt(1 + $side**2) - 1)/$side;
}
printf "%s=%f\n", $side, $side;
printf "pi=%f\n", $side*(2**($iter+2));
```

It prints (without so many line breaks)

```
[/ [- [sqrt [+ 1 [** [/ [- [sqrt [+ 1 [** [n 1] 2]]] 1]
                        [n 1]] 2]]] 1]
   [/ [- [sqrt [+ 1 [** [n 1] 2]]] 1] [n 1]]]=0.198912
pi=3.182598
```

The above module is very primitive. It does not implement mutator methods (++, -= and so on), does not do deep copying (not required without mutators!), and implements only those arithmetic operations which are used in the example.

To implement most arithmetic operattions is easy, one should just use the tables of operations, and change the code which fills `%subr` to

```
my %subr = ( 'n' => sub {$_[0]} );
foreach my $op (split " ", $overload::ops{with_assign}) {
  $subr{$op} = $subr{"$op="} = eval "sub {shift() $op shift()}";
}
my @bins = qw(binary 3way_comparison num_comparison str_comparison);
foreach my $op (split " ", "@overload::ops{ @bins }") {
  $subr{$op} = eval "sub {shift() $op shift()}";
}
foreach my $op (split " ", "@overload::ops{qw(unary func)}") {
  print "defining `$op'\n";
  $subr{$op} = eval "sub {$op shift()}";
}
```

Due to the section on *Calling Conventions for Mutators*, we do not need anything special to make += and friends work, except filling += entry of `%subr`, and defining a copy constructor (needed since Perl has no way to know that the implementation of `'+='` does not mutate the argument, compare the section on *Copy Constructor*).

To implement a copy constructor, add `'='` = \&cpy> to `use overload` line, and code (this code assumes that mutators change things one level deep only, so recursive copying is not needed):

```
sub cpy {
  my $self = shift;
  bless [@$self], ref $self;
}
```

To make ++ and -- work, we need to implement actual mutators, either directly, or in `nomethod`. We continue to do things inside `nomethod`, thus add

```
    if ($meth eq '++' or $meth eq '--') {
      @$obj = ($meth, (bless [@$obj]), 1); # Avoid circular reference
      return $obj;
    }
```

after the first line of *wrap()*. This is not a most effective implementation, one may consider

```
  sub inc { $_[0] = bless ['++', shift, 1]; }
```

instead.

As a final remark, note that one can fill `%subr` by

```
  my %subr = ( 'n' => sub {$_[0]} );
  foreach my $op (split " ", $overload::ops{with_assign}) {
    $subr{$op} = $subr{"$op="} = eval "sub {shift() $op shift()}";
  }
  my @bins = qw(binary 3way_comparison num_comparison str_comparison);
  foreach my $op (split " ", "@overload::ops{ @bins }") {
    $subr{$op} = eval "sub {shift() $op shift()}";
  }
  foreach my $op (split " ", "@overload::ops{qw(unary func)}") {
    $subr{$op} = eval "sub {$op shift()}";
  }
  $subr{'++'} = $subr{'+'};
  $subr{'--'} = $subr{'-'};
```

This finishes implementation of a primitive symbolic calculator in 50 lines of Perl code. Since the numeric values of subexpressions are not cached, the calculator is very slow.

Here is the answer for the exercise: In the case of *str()*, we need no explicit recursion since the overloaded `.`-operator will fall back to an existing overloaded operator `""`. Overloaded arithmetic operators *do not* fall back to numeric conversion if `fallback` is not explicitly requested. Thus without an explicit recursion *num()* would convert `['+', $a, $b]` to `$a + $b`, which would just rebuild the argument of *num()*.

If you wonder why defaults for conversion are different for *str()* and *num()*, note how easy it was to write the symbolic calculator. This simplicity is due to an appropriate choice of defaults. One extra note: due to teh explicit recursion *num()* is more fragile than *sym()*: we need to explicitly check for the type of `$a` and `$b`. If componets `$a` and `$b` happen to be of some related type, this may lead to problems.

*Really* symbolic calculator

One may wonder why we call the above calculator symbolic. The reason is that the actual calculation of the value of expression is postponed until the value is *used*.

To see it in action, add a method

```
  sub STORE {
    my $obj = shift;
    $#$obj = 1;
    @$obj->[0,1] = ('=', shift);
  }
```

to the package `symbolic`. After this change one can do

```
  my $a = new symbolic 3;
  my $b = new symbolic 4;
  my $c = sqrt($a**2 + $b**2);
```

and the numeric value of `$c` becomes 5. However, after calling

```
$a->STORE(12);  $b->STORE(5);
```

the numeric value of `$c` becomes 13. There is no doubt now that the module symbolic provides a *symbolic* calculator indeed.

To hide the rough edges under the hood, provide a *tie()*d interface to the package `symbolic` (compare with the section on *Metaphor clash*). Add methods

```
sub TIESCALAR { my $pack = shift; $pack->new(@_) }
sub FETCH { shift }
sub nop {  }          # Around a bug
```

(the bug is described in the section on *BUGS*). One can use this new interface as

```
tie $a, 'symbolic', 3;
tie $b, 'symbolic', 4;
$a->nop;  $b->nop;    # Around a bug

my $c = sqrt($a**2 + $b**2);
```

Now numeric value of `$c` is 5. After `$a = 12; $b = 5` the numeric value of `$c` becomes 13. To insulate the user of the module add a method

```
sub vars { my $p = shift; tie($_, $p), $_->nop foreach @_; }
```

Now

```
my ($a, $b);
symbolic->vars($a, $b);
my $c = sqrt($a**2 + $b**2);

$a = 3; $b = 4;
printf "c5  %s=%f\n", $c, $c;

$a = 12; $b = 5;
printf "c13  %s=%f\n", $c, $c;
```

shows that the numeric value of `$c` follows changes to the values of `$a` and `$b`.

**AUTHOR**

Ilya Zakharevich <*ilya@math.mps.ohio-state.edu*>.

**DIAGNOSTICS**

When Perl is run with the **–Do** switch or its equivalent, overloading induces diagnostic messages.

Using the `m` command of Perl debugger (see the *perldebug* manpage) one can deduce which operations are overloaded (and which ancestor triggers this overloading). Say, if `eq` is overloaded, then the method `(eq` is shown by debugger. The method `()` corresponds to the `fallback` key (in fact a presence of this method shows that this package has overloading enabled, and it is what is used by the `Overloaded` function of module `overload`).

**BUGS**

Because it is used for overloading, the per-package hash `%OVERLOAD` now has a special meaning in Perl. The symbol table is filled with names looking like line-noise.

For the purpose of inheritance every overloaded package behaves as if `fallback` is present (possibly undefined). This may create interesting effects if some package is not overloaded, but inherits from two overloaded packages.

Relation between overloading and *tie()*ing is broken. Overloading is triggered or not basing on the *previous* class of *tie()*d value.

This happens because the presence of overloading is checked too early, before any *tie()*d access is attempted. If the *FETCH()*ed class of the *tie()*d value does not change, a simple workaround is to access the value immediately after *tie()*ing, so that after this call the *previous* class coincides with the current one.

**Needed:** a way to fix this without a speed penalty.

Barewords are not covered by overloaded string constants.

This document is confusing. There are grammos and misleading language used in places. It would seem a total rewrite is needed.

## NAME

re – Perl pragma to alter regular expression behaviour

## SYNOPSIS

```
use re 'taint';
($x) = ($^X =~ /^(.*)$/s);     # $x is tainted here

$pat = '(?{ $foo = 1 })';
use re 'eval';
/foo${pat}bar/;                # won't fail (when not under -T switch)

{
    no re 'taint';             # the default
    ($x) = ($^X =~ /^(.*)$/s); # $x is not tainted here

    no re 'eval';              # the default
    /foo${pat}bar/;            # disallowed (with or without -T switch)
}

use re 'debug';                # NOT lexically scoped (as others are)
/^(.*)$/s;                     # output debugging info during
                               #     compile and run time

use re 'debugcolor';           # same as 'debug', but with colored output
...
```

(We use $^X in these examples because it's tainted by default.)

## DESCRIPTION

When `use re 'taint'` is in effect, and a tainted string is the target of a regex, the regex memories (or values returned by the m// operator in list context) are tainted. This feature is useful when regex operations on tainted data aren't meant to extract safe substrings, but to perform other transformations.

When `use re 'eval'` is in effect, a regex is allowed to contain `(?{ ... })` zero-width assertions even if regular expression contains variable interpolation. That is normally disallowed, since it is a potential security risk. Note that this pragma is ignored when the regular expression is obtained from tainted data, i.e. evaluation is always disallowed with tainted regular expresssions. See the section on *(?{ code })* in the *perlre* manpage.

For the purpose of this pragma, interpolation of precompiled regular expressions (i.e., the result of `qr//`) is *not* considered variable interpolation. Thus:

```
/foo${pat}bar/
```

*is* allowed if `$pat` is a precompiled regular expression, even if `$pat` contains `(?{ ... })` assertions.

When `use re 'debug'` is in effect, perl emits debugging messages when compiling and using regular expressions. The output is the same as that obtained by running a `-DDEBUGGING`–enabled perl interpreter with the **–Dr** switch. It may be quite voluminous depending on the complexity of the match. Using `debugcolor` instead of `debug` enables a form of output that can be used to get a colorful display on terminals that understand termcap color sequences. Set `$ENV{PERL_RE_TC}` to a comma-separated list of `termcap` properties to use for highlighting strings on/off, pre-point part on/off. See the section on *Debugging regular expressions* in the *perldebug* manpage for additional info.

The directive `use re 'debug'` is *not lexically scoped*, as the other directives are. It has both compile-time and run-time effects.

See the section on *Pragmatic Modules* in the *perlmodlib* manpage.

## NAME

sigtrap − Perl pragma to enable simple signal handling

## SYNOPSIS

```
use sigtrap;
use sigtrap qw(stack-trace old-interface-signals);  # equivalent
use sigtrap qw(BUS SEGV PIPE ABRT);
use sigtrap qw(die INT QUIT);
use sigtrap qw(die normal-signals);
use sigtrap qw(die untrapped normal-signals);
use sigtrap qw(die untrapped normal-signals
                stack-trace any error-signals);
use sigtrap 'handler' => \&my_handler, 'normal-signals';
use sigtrap qw(handler my_handler normal-signals
                stack-trace error-signals);
```

## DESCRIPTION

The **sigtrap** pragma is a simple interface to installing signal handlers. You can have it install one of two handlers supplied by **sigtrap** itself (one which provides a Perl stack trace and one which simply `die()`s), or alternately you can supply your own handler for it to install. It can be told only to install a handler for signals which are either untrapped or ignored. It has a couple of lists of signals to trap, plus you can supply your own list of signals.

The arguments passed to the `use` statement which invokes **sigtrap** are processed in order. When a signal name or the name of one of **sigtrap**'s signal lists is encountered a handler is immediately installed, when an option is encountered it affects subsequently installed handlers.

## OPTIONS

### SIGNAL HANDLERS

These options affect which handler will be used for subsequently installed signals.

**stack-trace**
The handler used for subsequently installed signals outputs a Perl stack trace to STDERR and then tries to dump core. This is the default signal handler.

**die** The handler used for subsequently installed signals calls `die` (actually `croak`) with a message indicating which signal was caught.

**handler** *your-handler*
*your-handler* will be used as the handler for subsequently installed signals. *your-handler* can be any value which is valid as an assignment to an element of `%SIG`.

### SIGNAL LISTS

**sigtrap** has a few built-in lists of signals to trap. They are:

**normal-signals**
These are the signals which a program might normally expect to encounter and which by default cause it to terminate. They are HUP, INT, PIPE and TERM.

**error-signals**
These signals usually indicate a serious problem with the Perl interpreter or with your script. They are ABRT, BUS, EMT, FPE, ILL, QUIT, SEGV, SYS and TRAP.

**old-interface-signals**
These are the signals which were trapped by default by the old **sigtrap** interface, they are ABRT, BUS, EMT, FPE, ILL, PIPE, QUIT, SEGV, SYS, TERM, and TRAP. If no signals or signals lists are passed to **sigtrap**, this list is used.

For each of these three lists, the collection of signals set to be trapped is checked before trapping; if your architecture does not implement a particular signal, it will not be trapped but rather silently ignored.

**OTHER**

**untrapped**
This token tells **sigtrap** to install handlers only for subsequently listed signals which aren't already trapped or ignored.

**any**
This token tells **sigtrap** to install handlers for all subsequently listed signals. This is the default behavior.

*signal*
Any argument which looks like a signal name (that is, `/^[A-Z][A-Z0-9]*$/`) indicates that **sigtrap** should install a handler for that name.

*number*
Require that at least version *number* of **sigtrap** is being used.

## EXAMPLES

Provide a stack trace for the old-interface-signals:

```
use sigtrap;
```

Ditto:

```
use sigtrap qw(stack-trace old-interface-signals);
```

Provide a stack trace on the 4 listed signals only:

```
use sigtrap qw(BUS SEGV PIPE ABRT);
```

Die on INT or QUIT:

```
use sigtrap qw(die INT QUIT);
```

Die on HUP, INT, PIPE or TERM:

```
use sigtrap qw(die normal-signals);
```

Die on HUP, INT, PIPE or TERM, except don't change the behavior for signals which are already trapped or ignored:

```
use sigtrap qw(die untrapped normal-signals);
```

Die on receipt one of an of the **normal-signals** which is currently **untrapped**, provide a stack trace on receipt of **any** of the **error-signals**:

```
use sigtrap qw(die untrapped normal-signals
               stack-trace any error-signals);
```

Install *my_handler()* as the handler for the **normal-signals**:

```
use sigtrap 'handler', \&my_handler, 'normal-signals';
```

Install *my_handler()* as the handler for the normal-signals, provide a Perl stack trace on receipt of one of the error-signals:

```
use sigtrap qw(handler my_handler normal-signals
               stack-trace error-signals);
```

**NAME**

strict – Perl pragma to restrict unsafe constructs

**SYNOPSIS**

```
use strict;

use strict "vars";
use strict "refs";
use strict "subs";

use strict;
no strict "vars";
```

**DESCRIPTION**

If no import list is supplied, all possible restrictions are assumed. (This is the safest mode to operate in, but is sometimes too strict for casual programming.) Currently, there are three possible things to be strict about: “subs”, “vars”, and “refs”.

`strict refs`

This generates a runtime error if you use symbolic references (see the *perlref* manpage).

```
use strict 'refs';
$ref = \$foo;
print $$ref;        # ok
$ref = "foo";
print $$ref;        # runtime error; normally ok
```

`strict vars`

This generates a compile-time error if you access a variable that wasn't declared via `use vars`, localized via `my()` or wasn't fully qualified. Because this is to avoid variable suicide problems and subtle dynamic scoping issues, a merely *local()* variable isn't good enough. See the `my` entry in the *perlfunc* manpage and the `local` entry in the *perlfunc* manpage.

```
use strict 'vars';
$X::foo = 1;         # ok, fully qualified
my $foo = 10;        # ok, my() var
local $foo = 9;      # blows up

package Cinna;
use vars qw/ $bar /;        # Declares $bar in current package
$bar = 'HgS';               # ok, global declared via pragma
```

The *local()* generated a compile-time error because you just touched a global name without fully qualifying it.

`strict subs`

This disables the poetry optimization, generating a compile-time error if you try to use a bareword identifier that's not a subroutine, unless it appears in curly braces or on the left hand side of the “=>” symbol.

```
use strict 'subs';
$SIG{PIPE} = Plumber;       # blows up
$SIG{PIPE} = "Plumber";     # just fine: bareword in curlies always ok
$SIG{PIPE} = \&Plumber;     # preferred form
```

See the section on *Pragmatic Modules* in the *perlmodlib* manpage.

## NAME

subs − Perl pragma to predeclare sub names

## SYNOPSIS

```
use subs qw(frob);
frob 3..10;
```

## DESCRIPTION

This will predeclare all the subroutine whose names are in the list, allowing you to use them without parentheses even before they're declared.

Unlike pragmas that affect the `$^H` hints variable, the `use vars` and `use subs` declarations are not BLOCK−scoped. They are thus effective for the entire file in which they appear. You may not rescind such declarations with `no vars` or `no subs`.

See the section on *Pragmatic Modules* in the *perlmodlib* manpage and the section on *strict subs* in the *strict* manpage.

**NAME**

vars − Perl pragma to predeclare global variable names

**SYNOPSIS**

```
use vars qw($frob @mung %seen);
```

**DESCRIPTION**

This will predeclare all the variables whose names are in the list, allowing you to use them under "use strict", and disabling any typo warnings.

Unlike pragmas that affect the `$^H` hints variable, the `use vars` and `use subs` declarations are not BLOCK−scoped. They are thus effective for the entire file in which they appear. You may not rescind such declarations with `no vars` or `no subs`.

Packages such as the **AutoLoader** and **SelfLoader** that delay loading of subroutines within packages can create problems with package lexicals defined using `my()`. While the **vars** pragma cannot duplicate the effect of package lexicals (total transparency outside of the package), it can act as an acceptable substitute by pre-declaring global symbols, ensuring their availability to the later-loaded routines.

See the section on *Pragmatic Modules* in the *perlmodlib* manpage.

**NAME**

AnyDBM_File − provide framework for multiple DBMs

NDBM_File, DB_File, GDBM_File, SDBM_File, ODBM_File − various DBM implementations

**SYNOPSIS**

```
use AnyDBM_File;
```

**DESCRIPTION**

This module is a ''pure virtual base class''— it has nothing of its own. It's just there to inherit from one of the various DBM packages. It prefers ndbm for compatibility reasons with Perl 4, then Berkeley DB (See the *DB_File* manpage), GDBM, SDBM (which is always there— it comes with Perl), and finally ODBM. This way old programs that used to use NDBM via *dbmopen()* can still do so, but new ones can reorder @ISA:

```
BEGIN { @AnyDBM_File::ISA = qw(DB_File GDBM_File NDBM_File) }
use AnyDBM_File;
```

Having multiple DBM implementations makes it trivial to copy database formats:

```
use POSIX; use NDBM_File; use DB_File;
tie %newhash,  'DB_File', $new_filename, O_CREAT|O_RDWR;
tie %oldhash,  'NDBM_File', $old_filename, 1, 0;
%newhash = %oldhash;
```

**DBM Comparisons**

Here's a partial table of features the different packages offer:

```
                         odbm    ndbm    sdbm    gdbm    bsd-db
                         ----    ----    ----    ----    ------
Linkage comes w/ perl    yes     yes     yes     yes     yes
Src comes w/ perl        no      no      yes     no      no
Comes w/ many unix os    yes     yes[0]  no      no      no
Builds ok on !unix       ?       ?       yes     yes     ?
Code Size                ?       ?       small   big     big
Database Size            ?       ?       small   big?    ok[1]
Speed                    ?       ?       slow    ok      fast
FTPable                  no      no      yes     yes     yes
Easy to build           N/A     N/A      yes     yes     ok[2]
Size limits              1k      4k      1k[3]   none    none
Byte-order independent   no      no      no      no      yes
Licensing restrictions   ?       ?       no      yes     no
```

[0] on mixed universe machines, may be in the bsd compat library, which is often shunned.

[1] Can be trimmed if you compile for one access method.

[2] See the *DB_File* manpage. Requires symbolic links.

[3] By default, but can be redefined.

**SEE ALSO**

*dbm* (3), *ndbm* (3), *DB_File* (3)

## NAME

AutoLoader – load subroutines only on demand

## SYNOPSIS

```
package Foo;
use AutoLoader 'AUTOLOAD';   # import the default AUTOLOAD subroutine

package Bar;
use AutoLoader;              # don't import AUTOLOAD, define our own
sub AUTOLOAD {
    ...
    $AutoLoader::AUTOLOAD = "...";
    goto &AutoLoader::AUTOLOAD;
}
```

## DESCRIPTION

The **AutoLoader** module works with the **AutoSplit** module and the `__END__` token to defer the loading of some subroutines until they are used rather than loading them all at once.

To use **AutoLoader**, the author of a module has to place the definitions of subroutines to be autoloaded after an `__END__` token. (See the *perldata* manpage.) The **AutoSplit** module can then be run manually to extract the definitions into individual files *auto/funcname.al*.

**AutoLoader** implements an AUTOLOAD subroutine. When an undefined subroutine in is called in a client module of **AutoLoader**, **AutoLoader**'s AUTOLOAD subroutine attempts to locate the subroutine in a file with a name related to the location of the file from which the client module was read. As an example, if *POSIX.pm* is located in */usr/local/lib/perl5/POSIX.pm*, **AutoLoader** will look for perl subroutines **POSIX** in */usr/local/lib/perl5/auto/POSIX/*.al*, where the `.al` file has the same name as the subroutine, sans package. If such a file exists, AUTOLOAD will read and evaluate it, thus (presumably) defining the needed subroutine. AUTOLOAD will then `goto` the newly defined subroutine.

Once this process completes for a given funtion, it is defined, so future calls to the subroutine will bypass the AUTOLOAD mechanism.

### Subroutine Stubs

In order for object method lookup and/or prototype checking to operate correctly even when methods have not yet been defined it is necessary to "forward declare" each subroutine (as in `sub NAME;`). See the section on *SYNOPSIS* in the *perlsub* manpage. Such forward declaration creates "subroutine stubs", which are place holders with no code.

The AutoSplit and **AutoLoader** modules automate the creation of forward declarations. The AutoSplit module creates an 'index' file containing forward declarations of all the AutoSplit subroutines. When the AutoLoader module is 'use'd it loads these declarations into its callers package.

Because of this mechanism it is important that **AutoLoader** is always `used` and not `required`.

### Using AutoLoader's AUTOLOAD Subroutine

In order to use **AutoLoader**'s AUTOLOAD subroutine you *must* explicitly import it:

```
use AutoLoader 'AUTOLOAD';
```

### Overriding AutoLoader's AUTOLOAD Subroutine

Some modules, mainly extensions, provide their own AUTOLOAD subroutines. They typically need to check for some special cases (such as constants) and then fallback to **AutoLoader**'s AUTOLOAD for the rest.

Such modules should *not* import **AutoLoader**'s AUTOLOAD subroutine. Instead, they should define their own AUTOLOAD subroutines along these lines:

```
use AutoLoader;
use Carp;

sub AUTOLOAD {
    my $constname;
    ($constname = $AUTOLOAD) =~ s/.*:://;
    my $val = constant($constname, @_ ? $_[0] : 0);
    if ($! != 0) {
        if ($! =~ /Invalid/) {
            $AutoLoader::AUTOLOAD = $AUTOLOAD;
            goto &AutoLoader::AUTOLOAD;
        }
        else {
            croak "Your vendor has not defined constant $constname";
        }
    }
    *$AUTOLOAD = sub { $val }; # same as: eval "sub $AUTOLOAD { $val }";
    goto &$AUTOLOAD;
}
```

If any module's own AUTOLOAD subroutine has no need to fallback to the AutoLoader's AUTOLOAD subroutine (because it doesn't have any AutoSplit subroutines), then that module should not use **AutoLoader** at all.

### Package Lexicals

Package lexicals declared with `my` in the main block of a package using **AutoLoader** will not be visible to auto-loaded subroutines, due to the fact that the given scope ends at the `__END__` marker. A module using such variables as package globals will not work properly under the **AutoLoader**.

The `vars` pragma (see the section on *vars* in the *perlmod* manpage) may be used in such situations as an alternative to explicitly qualifying all globals with the package namespace. Variables pre-declared with this pragma will be visible to any autoloaded routines (but will not be invisible outside the package, unfortunately).

### AutoLoader vs. SelfLoader

The **AutoLoader** is similar in purpose to **SelfLoader**: both delay the loading of subroutines.

**SelfLoader** uses the `__DATA__` marker rather than `__END__`. While this avoids the use of a hierarchy of disk files and the associated open/close for each routine loaded, **SelfLoader** suffers a startup speed disadvantage in the one-time parsing of the lines after `__DATA__`, after which routines are cached. **SelfLoader** can also handle multiple packages in a file.

**AutoLoader** only reads code as it is requested, and in many cases should be faster, but requires a machanism like **AutoSplit** be used to create the individual files. the *ExtUtils::MakeMaker* manpage will invoke **AutoSplit** automatically if **AutoLoader** is used in a module source file.

## CAVEATS

AutoLoaders prior to Perl 5.002 had a slightly different interface. Any old modules which use **AutoLoader** should be changed to the new calling style. Typically this just means changing a require to a use, adding the explicit `'AUTOLOAD'` import if needed, and removing **AutoLoader** from `@ISA`.

On systems with restrictions on file name length, the file corresponding to a subroutine may have a shorter name that the routine itself. This can lead to conflicting file names. The *AutoSplit* package warns of these potential conflicts when used to split a module.

AutoLoader may fail to find the autosplit files (or even find the wrong ones) in cases where `@INC` contains relative paths, **and** the program does `chdir`.

**SEE ALSO**

the *SelfLoader* manpage – an autoloader that doesn't use external files.

## NAME

AutoSplit – split a package for autoloading

## SYNOPSIS

```
autosplit($file, $dir, $keep, $check, $modtime);

autosplit_lib_modules(@modules);
```

## DESCRIPTION

This function will split up your program into files that the AutoLoader module can handle. It is used by both the standard perl libraries and by the MakeMaker utility, to automatically configure libraries for autoloading.

The `autosplit` interface splits the specified file into a hierarchy rooted at the directory `$dir`. It creates directories as needed to reflect class hierarchy, and creates the file *autosplit.ix*. This file acts as both forward declaration of all package routines, and as timestamp for the last update of the hierarchy.

The remaining three arguments to `autosplit` govern other options to the autosplitter.

$keep
: If the third argument, *$keep*, is false, then any pre-existing `*.al` files in the autoload directory are removed if they are no longer part of the module (obsoleted functions). `$keep` defaults to 0.

$check
: The fourth argument, *$check*, instructs `autosplit` to check the module currently being split to ensure that it does include a `use` specification for the AutoLoader module, and skips the module if AutoLoader is not detected. `$check` defaults to 1.

$modtime
: Lastly, the *$modtime* argument specifies that `autosplit` is to check the modification time of the module against that of the `autosplit.ix` file, and only split the module if it is newer. `$modtime` defaults to 1.

Typical use of AutoSplit in the perl MakeMaker utility is via the command-line with:

```
perl -e 'use AutoSplit; autosplit($ARGV[0], $ARGV[1], 0, 1, 1)'
```

Defined as a Make macro, it is invoked with file and directory arguments; `autosplit` will split the specified file into the specified directory and delete obsolete `.al` files, after checking first that the module does use the AutoLoader, and ensuring that the module is not already currently split in its current form (the modtime test).

The `autosplit_lib_modules` form is used in the building of perl. It takes as input a list of files (modules) that are assumed to reside in a directory **lib** relative to the current directory. Each file is sent to the autosplitter one at a time, to be split into the directory **lib/auto**.

In both usages of the autosplitter, only subroutines defined following the perl *__END__* token are split out into separate files. Some routines may be placed prior to this marker to force their immediate loading and parsing.

### Multiple packages

As of version 1.01 of the AutoSplit module it is possible to have multiple packages within a single file. Both of the following cases are supported:

```
package NAME;
__END__
sub AAA { ... }
package NAME::option1;
sub BBB { ... }
package NAME::option2;
sub BBB { ... }

package NAME;
__END__
sub AAA { ... }
sub NAME::option1::BBB { ... }
sub NAME::option2::BBB { ... }
```

**DIAGNOSTICS**

`AutoSplit` will inform the user if it is necessary to create the top-level directory specified in the invocation. It is preferred that the script or installation process that invokes `AutoSplit` have created the full directory path ahead of time. This warning may indicate that the module is being split into an incorrect path.

`AutoSplit` will warn the user of all subroutines whose name causes potential file naming conflicts on machines with drastically limited (8 characters or less) file name length. Since the subroutine name is used as the file name, these warnings can aid in portability to such systems.

Warnings are issued and the file skipped if `AutoSplit` cannot locate either the *__END__* marker or a "package Name;"–style specification.

`AutoSplit` will also emit general diagnostics for inability to create directories or files.

# NAME

B − The Perl Compiler

# SYNOPSIS

```
use B;
```

# DESCRIPTION

The `B` module supplies classes which allow a Perl program to delve into its own innards. It is the module used to implement the "backends" of the Perl compiler. Usage of the compiler does not require knowledge of this module: see the *O* module for the user-visible part. The `B` module is of use to those who want to write new compiler backends. This documentation assumes that the reader knows a fair amount about perl's internals including such things as SVs, OPs and the internal symbol table and syntax tree of a program.

# OVERVIEW OF CLASSES

The C structures used by Perl's internals to hold SV and OP information (PVIV, AV, HV, ..., OP, SVOP, UNOP, ...) are modelled on a class hierarchy and the `B` module gives access to them via a true object hierarchy. Structure fields which point to other objects (whether types of SV or types of OP) are represented by the `B` module as Perl objects of the appropriate class. The bulk of the `B` module is the methods for accessing fields of these structures. Note that all access is read-only: you cannot modify the internals by using this module.

## SV−RELATED CLASSES

B::IV, B::NV, B::RV, B::PV, B::PVIV, B::PVNV, B::PVMG, B::BM, B::PVLV, B::AV, B::HV, B::CV, B::GV, B::FM, B::IO. These classes correspond in the obvious way to the underlying C structures of similar names. The inheritance hierarchy mimics the underlying C "inheritance". Access methods correspond to the underlying C macros for field access, usually with the leading "class indication" prefix removed (Sv, Av, Hv, ...). The leading prefix is only left in cases where its removal would cause a clash in method name. For example, `GvREFCNT` stays as-is since its abbreviation would clash with the "superclass" method `REFCNT` (corresponding to the C function `SvREFCNT`).

## B::SV METHODS

REFCNT

FLAGS

## B::IV METHODS

IV

IVX

needs64bits

packiv

## B::NV METHODS

NV

NVX

## B::RV METHODS

RV

### B::PV METHODS

PV

### B::PVMG METHODS

MAGIC

SvSTASH

### B::MAGIC METHODS

MOREMAGIC

PRIVATE

TYPE

FLAGS

OBJ

PTR

### B::PVLV METHODS

TARGOFF

TARGLEN

TYPE

TARG

### B::BM METHODS

USEFUL

PREVIOUS

RARE

TABLE

### B::GV METHODS

NAME

STASH

SV

IO

FORM

AV

HV

EGV

CV

CVGEN

LINE

FILEGV

GvREFCNT

FLAGS

## B::IO METHODS

LINES

PAGE

PAGE_LEN

LINES_LEFT

TOP_NAME

TOP_GV

FMT_NAME

FMT_GV

BOTTOM_NAME

BOTTOM_GV

SUBPROCESS

IoTYPE

IoFLAGS

## B::AV METHODS

FILL

MAX

OFF

ARRAY

AvFLAGS

## B::CV METHODS

STASH

START

ROOT

GV

FILEGV

DEPTH

PADLIST

OUTSIDE

XSUB

XSUBANY

### B::HV METHODS

FILL

MAX

KEYS

RITER

NAME

PMROOT

ARRAY

### OP−RELATED CLASSES

B::OP, B::UNOP, B::BINOP, B::LOGOP, B::CONDOP, B::LISTOP, B::PMOP, B::SVOP, B::GVOP, B::PVOP, B::CVOP, B::LOOP, B::COP. These classes correspond in the obvious way to the underlying C structures of similar names. The inheritance hierarchy mimics the underlying C ‘‘inheritance’’. Access methods correspond to the underlying C structre field names, with the leading ‘‘class indication’’ prefix removed (op_).

### B::OP METHODS

next

sibling

ppaddr
This returns the function name as a string (e.g. pp_add, pp_rv2av).

desc
This returns the op description from the global C op_desc array (e.g. ‘‘addition" ‘‘array deref").

targ

type

seq

flags

private

### B::UNOP METHOD

first

### B::BINOP METHOD

last

### B::LOGOP METHOD

other

### B::CONDOP METHODS

true

false

### B::LISTOP METHOD

children

### B::PMOP METHODS

pmreplroot

pmreplstart

pmnext

pmregexp

pmflags

pmpermflags

precomp

### B::SVOP METHOD

sv

### B::GVOP METHOD

gv

### B::PVOP METHOD

pv

### B::LOOP METHODS

redoop

nextop

lastop

### B::COP METHODS

label

stash

filegv

cop_seq

arybase

line

## FUNCTIONS EXPORTED BY B

The B module exports a variety of functions: some are simple utility functions, others provide a Perl program with a way to get an initial "handle" on an internal object.

main_cv
: Return the (faked) CV corresponding to the main part of the Perl program.

main_root
: Returns the root op (i.e. an object in the appropriate B::OP–derived class) of the main part of the Perl program.

main_start
: Returns the starting op of the main part of the Perl program.

comppadlist
: Returns the AV object (i.e. in class B::AV) of the global comppadlist.

sv_undef
: Returns the SV object corresponding to the C variable `sv_undef`.

sv_yes
: Returns the SV object corresponding to the C variable `sv_yes`.

sv_no
: Returns the SV object corresponding to the C variable `sv_no`.

walkoptree(OP, METHOD)
: Does a tree-walk of the syntax tree based at OP and calls METHOD on each op it visits. Each node is visited before its children. If `walkoptree_debug` (q.v.) has been called to turn debugging on then the method `walkoptree_debug` is called on each op before METHOD is called.

walkoptree_debug(DEBUG)
: Returns the current debugging flag for `walkoptree`. If the optional DEBUG argument is non-zero, it sets the debugging flag to that. See the description of `walkoptree` above for what the debugging flag does.

walksymtable(SYMREF, METHOD, RECURSE)
: Walk the symbol table starting at SYMREF and call METHOD on each symbol visited. When the walk reached package symbols "Foo::" it invokes RECURSE and only recurses into the package if that sub returns true.

svref_2object(SV)
: Takes any Perl variable and turns it into an object in the appropriate B::OP−derived or B::SV−derived class. Apart from functions such as `main_root`, this is the primary way to get an initial "handle" on a internal perl data structure which can then be followed with the other access methods.

ppname(OPNUM)
: Return the PP function name (e.g. "pp_add") of op number OPNUM.

hash(STR)
: Returns a string in the form "0x..." representing the value of the internal hash function used by perl on string STR.

cast_I32(I)
: Casts I to the internal I32 type used by that perl.

minus_c
: Does the equivalent of the `−c` command-line option. Obviously, this is only useful in a BEGIN block or else the flag is set too late.

cstring(STR)
: Returns a double-quote-surrounded escaped version of STR which can be used as a string in C source code.

class(OBJ)
: Returns the class of an object without the part of the classname preceding the first "::". This is used to turn "B::UNOP" into "UNOP" for example.

threadsv_names
: In a perl compiled for threads, this returns a list of the special per-thread threadsv variables.

byteload_fh(FILEHANDLE)
: Load the contents of FILEHANDLE as bytecode. See documentation for the **Bytecode** module in *B::Backend* for how to generate bytecode.

**AUTHOR**

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

**NAME**

B::Asmdata – Autogenerated data about Perl ops, used to generate bytecode

**SYNOPSIS**

```
use Asmdata;
```

**DESCRIPTION**

See *ext/B/B/Asmdata.pm*.

**AUTHOR**

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

## NAME

B::Assembler − Assemble Perl bytecode

## SYNOPSIS

```
use Assembler;
```

## DESCRIPTION

See *ext/B/B/Assembler.pm*.

## AUTHOR

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

## NAME

B::Bblock – Walk basic blocks

## SYNOPSIS

```
perl -MO=Bblock[,OPTIONS] foo.pl
```

## DESCRIPTION

See *ext/B/README*.

## AUTHOR

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

## NAME

B::Bytecode – Perl compiler's bytecode backend

## SYNOPSIS

```
perl -MO=Bytecode[,OPTIONS] foo.pl
```

## DESCRIPTION

This compiler backend takes Perl source and generates a platform-independent bytecode encapsulating code to load the internal structures perl uses to run your program. When the generated bytecode is loaded in, your program is ready to run, reducing the time which perl would have taken to load and parse your program into its internal semi-compiled form. That means that compiling with this backend will not help improve the runtime execution speed of your program but may improve the start-up time. Depending on the environment in which your program runs this may or may not be a help.

The resulting bytecode can be run with a special byteperl executable or (for non-main programs) be loaded via the `byteload_fh` function in the *B* module.

## OPTIONS

If there are any non-option arguments, they are taken to be names of objects to be saved (probably doesn't work properly yet). Without extra arguments, it saves the main program.

**–ofilename**
Output to filename instead of STDOUT.

**--** Force end of options.

**–f** Force optimisations on or off one at a time. Each can be preceded by **no-** to turn the option off (e.g. **–fno-compress-nullops**).

**–fcompress-nullops**
Only fills in the necessary fields of ops which have been optimised away by perl's internal compiler.

**–fomit-sequence-numbers**
Leaves out code to fill in the op_seq field of all ops which is only used by perl's internal compiler.

**–fbypass-nullops**
If op->op_next ever points to a NULLOP, replaces the op_next field with the first non-NULLOP in the path of execution.

**–fstrip-syntax-tree**
Leaves out code to fill in the pointers which link the internal syntax tree together. They're not needed at run-time but leaving them out will make it impossible to recompile or disassemble the resulting program. It will also stop `goto label` statements from working.

**–On**
Optimisation level (n = 0, 1, 2, ...). **–O** means **–O1**. **–O1** sets **–fcompress-nullops –fomit-sequence numbers**. **–O6** adds **–fstrip-syntax-tree**.

**–D** Debug options (concatenated or separate flags like `perl -D`).

**–Do**
Prints each OP as it's processed.

**–Db**
Print debugging information about bytecompiler progress.

**–Da**
Tells the (bytecode) assembler to include source assembler lines in its output as bytecode comments.

**–DC**
Prints each CV taken from the final symbol tree walk.

**–S** Output (bytecode) assembler source rather than piping it through the assembler and outputting bytecode.

**–m** Compile as a module rather than a standalone program. Currently this just means that the bytecodes for initialising `main_start`, `main_root` and `curpad` are omitted.

**EXAMPLES**

```
perl –MO=Bytecode,–O6,–o,foo.plc foo.pl

perl –MO=Bytecode,–S foo.pl > foo.S
assemble foo.S > foo.plc
byteperl foo.plc

perl –MO=Bytecode,–m,–oFoo.pmc Foo.pm
```

**BUGS**

Plenty. Current status: experimental.

**AUTHOR**

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

## NAME

B::C – Perl compiler's C backend

## SYNOPSIS

```
perl -MO=C[,OPTIONS] foo.pl
```

## DESCRIPTION

This compiler backend takes Perl source and generates C source code corresponding to the internal structures that perl uses to run your program. When the generated C source is compiled and run, it cuts out the time which perl would have taken to load and parse your program into its internal semi-compiled form. That means that compiling with this backend will not help improve the runtime execution speed of your program but may improve the start-up time. Depending on the environment in which your program runs this may be either a help or a hindrance.

## OPTIONS

If there are any non-option arguments, they are taken to be names of objects to be saved (probably doesn't work properly yet). Without extra arguments, it saves the main program.

**–ofilename**
Output to filename instead of STDOUT

**–v** Verbose compilation (currently gives a few compilation statistics).

**--** Force end of options

**–uPackname**
Force apparently unused subs from package Packname to be compiled. This allows programs to use eval "*foo()*" even when sub foo is never seen to be used at compile time. The down side is that any subs which really are never used also have code generated. This option is necessary, for example, if you have a signal handler foo which you initialise with `$SIG{BAR} = "foo"`. A better fix, though, is just to change it to `$SIG{BAR} = \&foo`. You can have multiple **–u** options. The compiler tries to figure out which packages may possibly have subs in which need compiling but the current version doesn't do it very well. In particular, it is confused by nested packages (i.e. of the form `A::B`) where package `A` does not contain any subs.

**–D** Debug options (concatenated or separate flags like `perl -D`).

**–Do**
OPs, prints each OP as it's processed

**–Dc**
COPs, prints COPs as processed (incl. file & line num)

**–DA**
prints AV information on saving

**–DC**
prints CV information on saving

**–DM**
prints MAGIC information on saving

**–f** Force optimisations on or off one at a time.

**–fcog**
Copy-on-grow: PVs declared and initialised statically.

**–fno-cog**
No copy-on-grow.

**–On**
Optimisation level (n = 0, 1, 2, ...). **–O** means **–O1**. Currently, **–O1** and higher set **–fcog**.

**EXAMPLES**

```
perl -MO=C,-ofoo.c foo.pl
perl cc_harness -o foo foo.c
```

Note that `cc_harness` lives in the `B` subdirectory of your perl library directory. The utility called `perlcc` may also be used to help make use of this compiler.

```
perl -MO=C,-v,-DcA bar.pl > /dev/null
```

**BUGS**

Plenty. Current status: experimental.

**AUTHOR**

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

## NAME

B::CC – Perl compiler's optimized C translation backend

## SYNOPSIS

```
perl -MO=CC[,OPTIONS] foo.pl
```

## DESCRIPTION

This compiler backend takes Perl source and generates C source code corresponding to the flow of your program. In other words, this backend is somewhat a "real" compiler in the sense that many people think about compilers. Note however that, currently, it is a very poor compiler in that although it generates (mostly, or at least sometimes) correct code, it performs relatively few optimisations. This will change as the compiler develops. The result is that running an executable compiled with this backend may start up more quickly than running the original Perl program (a feature shared by the **C** compiler backend— see *B::C*) and may also execute slightly faster. This is by no means a good optimising compiler— yet.

## OPTIONS

If there are any non-option arguments, they are taken to be names of objects to be saved (probably doesn't work properly yet). Without extra arguments, it saves the main program.

**–ofilename**
Output to filename instead of STDOUT

**–v** Verbose compilation (currently gives a few compilation statistics).

**--** Force end of options

**–uPackname**
Force apparently unused subs from package Packname to be compiled. This allows programs to use eval "*foo()*" even when sub foo is never seen to be used at compile time. The down side is that any subs which really are never used also have code generated. This option is necessary, for example, if you have a signal handler foo which you initialise with `$SIG{BAR} = "foo"`. A better fix, though, is just to change it to `$SIG{BAR} = \&foo`. You can have multiple **–u** options. The compiler tries to figure out which packages may possibly have subs in which need compiling but the current version doesn't do it very well. In particular, it is confused by nested packages (i.e. of the form `A::B`) where package `A` does not contain any subs.

**–mModulename**
Instead of generating source for a runnable executable, generate source for an XSUB module. The boot_Modulename function (which DynaLoader can look for) does the appropriate initialisation and runs the main part of the Perl source that is being compiled.

**–D** Debug options (concatenated or separate flags like `perl -D`).

**–Dr**
Writes debugging output to STDERR just as it's about to write to the program's runtime (otherwise writes debugging info as comments in its C output).

**–DO**
Outputs each OP as it's compiled

**–Ds**
Outputs the contents of the shadow stack at each OP

**–Dp**
Outputs the contents of the shadow pad of lexicals as it's loaded for each sub or the main program.

**–Dq**
Outputs the name of each fake PP function in the queue as it's about to process it.

**–Dl**
Output the filename and line number of each original line of Perl code as it's processed (`pp_nextstate`).

**–Dt**
Outputs timing information of compilation stages.

**–f** Force optimisations on or off one at a time.

**–ffreetmps-each-bblock**
Delays FREETMPS from the end of each statement to the end of the each basic block.

**–ffreetmps-each-loop**
Delays FREETMPS from the end of each statement to the end of the group of basic blocks forming a loop. At most one of the freetmps-each-* options can be used.

**–fomit-taint**
Omits generating code for handling perl's tainting mechanism.

**–On**
Optimisation level (n = 0, 1, 2, ...). **–O** means **–O1**. Currently, **–O1** sets **–ffreetmps-each-bblock** and **–O2** sets **–ffreetmps-each-loop**.

## EXAMPLES

```
perl -MO=CC,-O2,-ofoo.c foo.pl
perl cc_harness -o foo foo.c
```

Note that `cc_harness` lives in the `B` subdirectory of your perl library directory. The utility called `perlcc` may also be used to help make use of this compiler.

```
perl -MO=CC,-mFoo,-oFoo.c Foo.pm
perl cc_harness -shared -c -o Foo.so Foo.c
```

## BUGS

Plenty. Current status: experimental.

## DIFFERENCES

These aren't really bugs but they are constructs which are heavily tied to perl's compile-and-go implementation and with which this compiler backend cannot cope.

### Loops

Standard perl calculates the target of "next", "last", and "redo" at run-time. The compiler calculates the targets at compile-time. For example, the program

```
sub skip_on_odd { next NUMBER if $_[0] % 2 }
NUMBER: for ($i = 0; $i < 5; $i++) {
    skip_on_odd($i);
    print $i;
}
```

produces the output

```
024
```

with standard perl but gives a compile-time error with the compiler.

### Context of ".."

The context (scalar or array) of the ".." operator determines whether it behaves as a range or a flip/flop. Standard perl delays until runtime the decision of which context it is in but the compiler needs to know the context at compile-time. For example,

```
@a = (4,6,1,0,0,1);
sub range { (shift @a)..(shift @a) }
print range();
while (@a) { print scalar(range()) }
```

generates the output

```
456123E0
```

with standard Perl but gives a compile-time error with compiled Perl.

### Arithmetic

Compiled Perl programs use native C arithemtic much more frequently than standard perl. Operations on large numbers or on boundary cases may produce different behaviour.

### Deprecated features

Features of standard perl such as `$[` which have been deprecated in standard perl since Perl5 was released have not been implemented in the compiler.

## AUTHOR

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

**NAME**

B::Debug – Walk Perl syntax tree, printing debug info about ops

**SYNOPSIS**

```
perl -MO=Debug[,OPTIONS] foo.pl
```

**DESCRIPTION**

See *ext/B/README*.

**AUTHOR**

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

## NAME

B::Deparse – Perl compiler backend to produce perl code

## SYNOPSIS

**perl –MO=Deparse**[**,–u***PACKAGE*][**,–p**][**,–l**][**,–s***LETTERS*] *prog.pl*

## DESCRIPTION

B::Deparse is a backend module for the Perl compiler that generates perl source code, based on the internal compiled structure that perl itself creates after parsing a program. The output of B::Deparse won't be exactly the same as the original source, since perl doesn't keep track of comments or whitespace, and there isn't a one-to-one correspondence between perl's syntactical constructions and their compiled form, but it will often be close. When you use the **–p** option, the output also includes parentheses even when they are not required by precedence, which can make it easy to see if perl is parsing your expressions the way you intended.

Please note that this module is mainly new and untested code and is still under development, so it may change in the future.

## OPTIONS

As with all compiler backend options, these must follow directly after the '–MO=Deparse', separated by a comma but not any white space.

**–p** Print extra parentheses. Without this option, B::Deparse includes parentheses in its output only when they are needed, based on the structure of your program. With **–p**, it uses parentheses (almost) whenever they would be legal. This can be useful if you are used to LISP, or if you want to see how perl parses your input. If you say

```
if ($var & 0x7f == 65) {print "Gimme an A!"}
print ($which ? $a : $b), "\n";
$name = $ENV{USER} or "Bob";
```

`B::Deparse,-p` will print

```
if (($var & 0)) {
    print('Gimme an A!')
};
(print(($which ? $a : $b)), '???');
(($name = $ENV{'USER'}) or '???')
```

which probably isn't what you intended (the `'???'` is a sign that perl optimized away a constant value).

**–u***PACKAGE*

Normally, B::Deparse deparses the main code of a program, all the subs called by the main program (and all the subs called by them, recursively), and any other subs in the main:: package. To include subs in other packages that aren't called directly, such as AUTOLOAD, DESTROY, other subs called automatically by perl, and methods, which aren't resolved to subs until runtime, use the **–u** option. The argument to **–u** is the name of a package, and should follow directly after the 'u'. Multiple **–u** options may be given, separated by commas. Note that unlike some other backends, B::Deparse doesn't (yet) try to guess automatically when **–u** is needed -- you must invoke it yourself.

**–l** Add '#line' declarations to the output based on the line and file locations of the original code.

**–s***LETTERS*

Tweak the style of B::Deparse's output. At the moment, only one style option is implemented:

**C** Cuddle `elsif`, `else`, and `continue` blocks. For example, print

```
if (...) {
     ...
} else {
     ...
}
```

instead of

```
if (...) {
     ...
}
else {
     ...
}
```

The default is not to cuddle.

**BUGS**

See the ‘to do’ list at the beginning of the module file.

**AUTHOR**

Stephen McCamant <alias@mcs.com>, based on an earlier version by Malcolm Beattie <mbeattie@sable.ox.ac.uk>.

## NAME

B::Disassembler − Disassemble Perl bytecode

## SYNOPSIS

```
use Disassembler;
```

## DESCRIPTION

See *ext/B/B/Disassembler.pm*.

## AUTHOR

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

## NAME

B::Lint – Perl lint

## SYNOPSIS

perl –MO=Lint[,OPTIONS] foo.pl

## DESCRIPTION

The B::Lint module is equivalent to an extended version of the **–w** option of **perl**. It is named after the program **lint** which carries out a similar process for C programs.

## OPTIONS AND LINT CHECKS

Option words are separated by commas (not whitespace) and follow the usual conventions of compiler backend options. Following any options (indicated by a leading **–**) come lint check arguments. Each such argument (apart from the special **all** and **none** options) is a word representing one possible lint check (turning on that check) or is **no-foo** (turning off that check). Before processing the check arguments, a standard list of checks is turned on. Later options override earlier ones. Available options are:

**context** Produces a warning whenever an array is used in an implicit scalar context. For example, both of the lines

```
    $foo = length(@bar);
    $foo = @bar;
will elicit a warning. Using an explicit B<scalar()> silences the
warning. For example,

    $foo = scalar(@bar);
```

**implicit-read** and **implicit-write**

These options produce a warning whenever an operation implicitly reads or (respectively) writes to one of Perl's special variables. For example, **implicit-read** will warn about these:

```
/foo/;
```

and **implicit-write** will warn about these:

```
s/foo/bar/;
```

Both **implicit-read** and **implicit-write** warn about this:

```
for (@a) { ... }
```

**dollar-underscore**

This option warns whenever `$_` is used either explicitly anywhere or as the implicit argument of a **print** statement.

**private-names**

This option warns on each use of any variable, subroutine or method name that lives in a non-current package but begins with an underscore ("_"). Warnings aren't issued for the special case of the single character name "_" by itself (e.g. `$_` and `@_`).

**undefined-subs**

This option warns whenever an undefined subroutine is invoked. This option will only catch explicitly invoked subroutines such as `foo()` and not indirect invocations such as `&$subref()` or `$obj->meth()`. Note that some programs or modules delay definition of subs until runtime by means of the AUTOLOAD mechanism.

**regexp-variables**

This option warns whenever one of the regexp variables $’, $& or $’ is used. Any occurrence of any of these variables in your program can slow your whole program down. See the *perlre* manpage for details.

**all** Turn all warnings on.

**none** Turn all warnings off.

## NON LINT−CHECK OPTIONS

**−u Package**

Normally, Lint only checks the main code of the program together with all subs defined in package main. The **−u** option lets you include other package names whose subs are then checked by Lint.

## BUGS

This is only a very preliminary version.

## AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk.

## NAME

B::Showlex − Show lexical variables used in functions or files

## SYNOPSIS

```
perl −MO=Showlex[,SUBROUTINE] foo.pl
```

## DESCRIPTION

When a subroutine name is provided in OPTIONS, prints the lexical variables used in that subroutine. Otherwise, prints the file-scope lexicals in the file.

## AUTHOR

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

**NAME**

B::Stackobj − Helper module for CC backend

**SYNOPSIS**

```
use B::Stackobj;
```

**DESCRIPTION**

See *ext/B/README*.

**AUTHOR**

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

## NAME

B::Terse − Walk Perl syntax tree, printing terse info about ops

## SYNOPSIS

```
perl -MO=Terse[,OPTIONS] foo.pl
```

## DESCRIPTION

See *ext/B/README*.

## AUTHOR

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

## NAME

B::Xref − Generates cross reference reports for Perl programs

## SYNOPSIS

perl −MO=Xref[,OPTIONS] foo.pl

## DESCRIPTION

The B::Xref module is used to generate a cross reference listing of all definitions and uses of variables, subroutines and formats in a Perl program. It is implemented as a backend for the Perl compiler.

The report generated is in the following format:

```
File filename1
  Subroutine subname1
    Package package1
      object1        C<line numbers>
      object2        C<line numbers>
      ...
    Package package2
    ...
```

Each **File** section reports on a single file. Each **Subroutine** section reports on a single subroutine apart from the special cases "(definitions)" and "(main)". These report, respectively, on subroutine definitions found by the initial symbol table walk and on the main part of the program or module external to all subroutines.

The report is then grouped by the **Package** of each variable, subroutine or format with the special case "(lexicals)" meaning lexical variables. Each **object** name (implicitly qualified by its containing **Package**) includes its type *character* (s) at the beginning where possible. Lexical variables are easier to track and even included dereferencing information where possible.

The `line numbers` are a comma separated list of line numbers (some preceded by code letters) where that object is used in some way. Simple uses aren't preceded by a code letter. Introductions (such as where a lexical is first defined with `my`) are indicated with the letter "i". Subroutine and method calls are indicated by the character "&". Subroutine definitions are indicated by "s" and format definitions by "f".

## OPTIONS

Option words are separated by commas (not whitespace) and follow the usual conventions of compiler backend options.

`−oFILENAME`
: Directs output to `FILENAME` instead of standard output.

`−r`
: Raw output. Instead of producing a human-readable report, outputs a line in machine-readable form for each definition/use of a variable/sub/format.

`−D[tO]`
: (Internal) debug options, probably only useful if `−r` included. The `t` option prints the object on the top of the stack as it's being tracked. The `O` option prints each operator as it's being processed in the execution order of the program.

## BUGS

Non-lexical variables are quite difficult to track through a program. Sometimes the type of a non-lexical variable's use is impossible to determine. Introductions of non-lexical non-scalars don't seem to be reported properly.

## AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk.

**NAME**

Benchmark − benchmark running times of code

timethis − run a chunk of code several times

timethese − run several chunks of code several times

timeit − run a chunk of code and see how long it goes

**SYNOPSIS**

```
timethis ($count, "code");

# Use Perl code in strings...
timethese($count, {
    'Name1' => '...code1...',
    'Name2' => '...code2...',
});

# ... or use subroutine references.
timethese($count, {
    'Name1' => sub { ...code1... },
    'Name2' => sub { ...code2... },
});

$t = timeit($count, '...other code...')
print "$count loops of other code took:",timestr($t),"\n";
```

**DESCRIPTION**

The Benchmark module encapsulates a number of routines to help you figure out how long it takes to execute some code.

### Methods

new Returns the current time. Example:

```
use Benchmark;
$t0 = new Benchmark;
# ... your code here ...
$t1 = new Benchmark;
$td = timediff($t1, $t0);
print "the code took:",timestr($td),"\n";
```

debug Enables or disable debugging by setting the `$Benchmark::Debug` flag:

```
debug Benchmark 1;
$t = timeit(10, ' 5 ** $Global ');
debug Benchmark 0;
```

### Standard Exports

The following routines will be exported into your namespace if you use the Benchmark module:

timeit(COUNT, CODE)

Arguments: COUNT is the number of times to run the loop, and CODE is the code to run. CODE may be either a code reference or a string to be eval'd; either way it will be run in the caller's package.

Returns: a Benchmark object.

timethis ( COUNT, CODE, [ TITLE, [ STYLE ]] )
Time COUNT iterations of CODE. CODE may be a string to eval or a code reference; either way the CODE will run in the caller's package. Results will be printed to STDOUT as TITLE followed by the times. TITLE defaults to "timethis COUNT" if none is provided. STYLE determines the format of the output, as described for *timestr()* below.

The COUNT can be zero or negative: this means the *minimum number of CPU seconds* to run. A zero signifies the default of 3 seconds. For example to run at least for 10 seconds:

```
timethis(-10, $code)
```

or to run two pieces of code tests for at least 3 seconds:

```
timethese(0, { test1 => '...', test2 => '...'})
```

CPU seconds is, in UNIX terms, the user time plus the system time of the process itself, as opposed to the real (wallclock) time and the time spent by the child processes. Less than 0.1 seconds is not accepted (−0.01 as the count, for example, will cause a fatal runtime exception).

Note that the CPU seconds is the **minimum** time: CPU scheduling and other operating system factors may complicate the attempt so that a little bit more time is spent. The benchmark output will, however, also tell the number of `$code` runs/second, which should be a more interesting number than the actually spent seconds.

Returns a Benchmark object.

timethese ( COUNT, CODEHASHREF, [ STYLE ] )
The CODEHASHREF is a reference to a hash containing names as keys and either a string to eval or a code reference for each value. For each (KEY, VALUE) pair in the CODEHASHREF, this routine will call

```
timethis(COUNT, VALUE, KEY, STYLE)
```

The routines are called in string comparison order of KEY.

The COUNT can be zero or negative, see *timethis()*.

timediff ( T1, T2 )
Returns the difference between two Benchmark times as a Benchmark object suitable for passing to *timestr()*.

timestr ( TIMEDIFF, [ STYLE, [ FORMAT ] ] )
Returns a string that formats the times in the TIMEDIFF object in the requested STYLE. TIMEDIFF is expected to be a Benchmark object similar to that returned by *timediff()*.

STYLE can be any of 'all', 'noc', 'nop' or 'auto'. 'all' shows each of the 5 times available ('wallclock' time, user time, system time, user time of children, and system time of children). 'noc' shows all except the two children times. 'nop' shows only wallclock and the two children times. 'auto' (the default) will act as 'all' unless the children times are both zero, in which case it acts as 'noc'.

FORMAT is the the *printf(3)* manpage-style format specifier (without the leading '%') to use to print the times. It defaults to '5.2f'.

### Optional Exports

The following routines will be exported into your namespace if you specifically ask that they be imported:

clearcache ( COUNT )
Clear the cached time for COUNT rounds of the null loop.

clearallcache ( )
: Clear all cached times.

disablecache ( )
: Disable caching of timings for the null loop. This will force Benchmark to recalculate these timings for each new piece of code timed.

enablecache ( )
: Enable caching of timings for the null loop. The time taken for COUNT rounds of the null loop will be calculated only once for each different COUNT used.

**NOTES**

The data is stored as a list of values from the time and times functions:

```
    ($real, $user, $system, $children_user, $children_system)
```

in seconds for the whole loop (not divided by the number of rounds).

The timing is done using *time* (3) and *times* (3).

Code is executed in the caller's package.

The time of the null loop (a loop with the same number of rounds but empty loop body) is subtracted from the time of the real loop.

The null loop times are cached, the key being the number of rounds. The caching can be controlled using calls like these:

```
clearcache($key);
clearallcache();

disablecache();
enablecache();
```

**INHERITANCE**

Benchmark inherits from no other class, except of course for Exporter.

**CAVEATS**

Comparing eval'd strings with code references will give you inaccurate results: a code reference will show a slower execution time than the equivalent eval'd string.

The real time timing is done using *time* (2) and the granularity is therefore only one second.

Short tests may produce negative figures because perl can appear to take longer to execute the empty loop than a short test; try:

```
timethis(100,'1');
```

The system time of the null loop might be slightly more than the system time of the loop with the actual code and therefore the difference might end up being < 0.

**AUTHORS**

Jarkko Hietaniemi <*jhi@iki.fi*>, Tim Bunce <*Tim.Bunce@ig.co.uk*>

**MODIFICATION HISTORY**

September 8th, 1994; by Tim Bunce.

March 28th, 1997; by Hugo van der Sanden: added support for code references and the already documented 'debug' method; revamped documentation.

April 04-07th, 1997: by Jarkko Hietaniemi, added the run-for-some-time functionality.

**NAME**

carp – warn of errors (from perspective of caller)

cluck – warn of errors with stack backtrace
(not exported by default)

croak – die of errors (from perspective of caller)

confess – die of errors with stack backtrace

**SYNOPSIS**

```
use Carp;
croak "We're outta here!";

use Carp qw(cluck);
cluck "This is how we got here!";
```

**DESCRIPTION**

The Carp routines are useful in your own modules because they act like *die()* or *warn()*, but report where the error was in the code they were called from. Thus if you have a routine *Foo()* that has a *carp()* in it, then the *carp()* will report the error as occurring where *Foo()* was called, not where *carp()* was called.

### Forcing a Stack Trace

As a debugging aid, you can force Carp to treat a croak as a confess and a carp as a cluck across *all* modules. In other words, force a detailed stack trace to be given. This can be very helpful when trying to understand why, or from where, a warning or error is being generated.

This feature is enabled by 'importing' the non-existant symbol 'verbose'. You would typically enable it by saying

```
perl -MCarp=verbose script.pl
```

or by including the string `MCarp=verbose` in the the *PERL5OPT* manpage environment variable.

**NAME**

CGI − Simple Common Gateway Interface Class

**SYNOPSIS**

```
# CGI script that creates a fill-out form
# and echoes back its values.

use CGI qw/:standard/;
print header,
      start_html('A Simple Example'),
      h1('A Simple Example'),
      start_form,
      "What's your name? ",textfield('name'),p,
      "What's the combination?", p,
      checkbox_group(-name=>'words',
                     -values=>['eenie','meenie','minie','moe'],
                     -defaults=>['eenie','minie']), p,
      "What's your favorite color? ",
      popup_menu(-name=>'color',
                 -values=>['red','green','blue','chartreuse']),p,
      submit,
      end_form,
      hr;

 if (param()) {
     print "Your name is",em(param('name')),p,
           "The keywords are: ",em(join(", ",param('words'))),p,
           "Your favorite color is ",em(param('color')),
           hr;
 }
```

**ABSTRACT**

This perl library uses perl5 objects to make it easy to create Web fill-out forms and parse their contents. This package defines CGI objects, entities that contain the values of the current query string and other state variables. Using a CGI object's methods, you can examine keywords and parameters passed to your script, and create forms whose initial values are taken from the current query (thereby preserving state information). The module provides shortcut functions that produce boilerplate HTML, reducing typing and coding errors. It also provides functionality for some of the more advanced features of CGI scripting, including support for file uploads, cookies, cascading style sheets, server push, and frames.

CGI.pm also provides a simple function-oriented programming style for those who don't need its object-oriented features.

The current version of CGI.pm is available at

```
http://www.genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.html
ftp://ftp-genome.wi.mit.edu/pub/software/WWW/
```


**DESCRIPTION**

**PROGRAMMING STYLE**

There are two styles of programming with CGI.pm, an object-oriented style and a function-oriented style. In the object-oriented style you create one or more CGI objects and then use object methods to create the various elements of the page. Each CGI object starts out with the list of named parameters that were passed to your CGI script by the server. You can modify the objects, save them to a file or database and recreate them. Because each object corresponds to the "state" of the CGI script, and because each object's parameter list is independent of the others, this allows you to save the state of the script and restore it later.

For example, using the object oriented style, here is now you create a simple "Hello World" HTML page:

```
#!/usr/local/bin/pelr
use CGI;                             # load CGI routines
$q = new CGI;                        # create new CGI object
print $q->header,                    # create the HTTP header
      $q->start_html('hello world'), # start the HTML
      $q->h1('hello world'),         # level 1 header
      $q->end_html;                  # end the HTML
```

In the function-oriented style, there is one default CGI object that you rarely deal with directly. Instead you just call functions to retrieve CGI parameters, create HTML tags, manage cookies, and so on. This provides you with a cleaner programming interface, but limits you to using one CGI object at a time. The following example prints the same page, but uses the function-oriented interface. The main differences are that we now need to import a set of functions into our name space (usually the "standard" functions), and we don't need to create the CGI object.

```
#!/usr/local/bin/pelr
use CGI qw/:standard/;           # load standard CGI routines
print header,                    # create the HTTP header
      start_html('hello world'), # start the HTML
      h1('hello world'),         # level 1 header
      end_html;                  # end the HTML
```

The examples in this document mainly use the object-oriented style. See HOW TO IMPORT FUNCTIONS for important information on function-oriented programming in CGI.pm

**CALLING CGI.PM ROUTINES**

Most CGI.pm routines accept several arguments, sometimes as many as 20 optional ones! To simplify this interface, all routines use a named argument calling style that looks like this:

```
print $q->header(-type=>'image/gif',-expires=>'+3d');
```

Each argument name is preceded by a dash. Neither case nor order matters in the argument list. −type, −Type, and −TYPE are all acceptable. In fact, only the first argument needs to begin with a dash. If a dash is present in the first argument, CGI.pm assumes dashes for the subsequent ones.

You don't have to use the hyphen at allif you don't want to. After creating a CGI object, call the **use_named_parameters()** method with a nonzero value. This will tell CGI.pm that you intend to use named parameters exclusively:

```
$query = new CGI;
$query->use_named_parameters(1);
$field = $query->radio_group('name'=>'OS',
                             'values'=>['Unix','Windows','Macintosh'],
                             'default'=>'Unix');
```

Several routines are commonly called with just one argument. In the case of these routines you can provide the single argument without an argument name. *header()* happens to be one of these routines. In this case, the single argument is the document type.

```
print $q->header('text/html');
```

Other such routines are documented below.

Sometimes named arguments expect a scalar, sometimes a reference to an array, and sometimes a reference to a hash. Often, you can pass any type of argument and the routine will do whatever is most appropriate. For example, the *param()* routine is used to set a CGI parameter to a single or a multi-valued value. The two cases are shown below:

```
$q->param(-name=>'veggie',-value=>'tomato');
$q->param(-name=>'veggie',-value=>'[tomato','tomahto','potato','potahto']);
```

A large number of routines in CGI.pm actually aren't specifically defined in the module, but are generated automatically as needed. These are the "HTML shortcuts," routines that generate HTML tags for use in dynamically-generated pages. HTML tags have both attributes (the attribute="value" pairs within the tag itself) and contents (the part between the opening and closing pairs.) To distinguish between attributes and contents, CGI.pm uses the convention of passing HTML attributes as a hash reference as the first argument, and the contents, if any, as any subsequent arguments. It works out like this:

```
Code                           Generated HTML
----                           --------------
h1()                           <H1>
h1('some','contents');         <H1>some contents</H1>
h1({-align=>left});            <H1 ALIGN="LEFT">
h1({-align=>left},'contents'); <H1 ALIGN="LEFT">contents</H1>
```

HTML tags are described in more detail later.

Many newcomers to CGI.pm are puzzled by the difference between the calling conventions for the HTML shortcuts, which require curly braces around the HTML tag attributes, and the calling conventions for other routines, which manage to generate attributes without the curly brackets. Don't be confused. As a convenience the curly braces are optional in all but the HTML shortcuts. If you like, you can use curly braces when calling any routine that takes named arguments. For example:

```
print $q->header( {-type=>'image/gif',-expires=>'+3d'} );
```

If you use the **−w** switch, you will be warned that some CGI.pm argument names conflict with built-in Perl functions. The most frequent of these is the −values argument, used to create multi-valued menus, radio button clusters and the like. To get around this warning, you have several choices:

1. Use another name for the argument, if one is available. For example, −value is an alias for −values.

2. Change the capitalization, e.g. −Values

3. Put quotes around the argument name, e.g. '−values'

Many routines will do something useful with a named argument that it doesn't recognize. For example, you can produce non-standard HTTP header fields by providing them as named arguments:

```
print $q->header(-type  =>  'text/html',
                 -cost  =>  'Three smackers',
                 -annoyance_level => 'high',
                 -complaints_to   => 'bit bucket');
```

This will produce the following nonstandard HTTP header:

```
HTTP/1.0 200 OK
Cost: Three smackers
Annoyance-level: high
Complaints-to: bit bucket
Content-type: text/html
```

Notice the way that underscores are translated automatically into hyphens. HTML−generating routines perform a different type of translation.

This feature allows you to keep up with the rapidly changing HTTP and HTML "standards".

**CREATING A NEW QUERY OBJECT (OBJECT−ORIENTED STYLE):**

```
$query = new CGI;
```

This will parse the input (from both POST and GET methods) and store it into a perl5 object called `$query`.

**CREATING A NEW QUERY OBJECT FROM AN INPUT FILE**

```
$query = new CGI(INPUTFILE);
```

If you provide a file handle to the *new()* method, it will read parameters from the file (or STDIN, or whatever). The file can be in any of the forms describing below under debugging (i.e. a series of newline delimited TAG=VALUE pairs will work). Conveniently, this type of file is created by the *save()* method (see below). Multiple records can be saved and restored.

Perl purists will be pleased to know that this syntax accepts references to file handles, or even references to filehandle globs, which is the "official" way to pass a filehandle:

```
$query = new CGI(\*STDIN);
```

You can also initialize the CGI object with a FileHandle or IO::File object.

If you are using the function-oriented interface and want to initialize CGI state from a file handle, the way to do this is with **restore_parameters()**. This will (re)initialize the default CGI object from the indicated file handle.

```
open (IN,"test.in") || die;
restore_parameters(IN);
close IN;
```

You can also initialize the query object from an associative array reference:

```
$query = new CGI( {'dinosaur'=>'barney',
                   'song'=>'I love you',
                   'friends'=>[qw/Jessica George Nancy/]}
                );
```

or from a properly formatted, URL−escaped query string:

```
$query = new CGI('dinosaur=barney&color=purple');
```

or from a previously existing CGI object (currently this clones the parameter list, but none of the other object-specific fields, such as autoescaping):

```
$old_query = new CGI;
$new_query = new CGI($old_query);
```

To create an empty query, initialize it from an empty string or hash:

```
$empty_query = new CGI("");

    -or-

$empty_query = new CGI({});
```

**FETCHING A LIST OF KEYWORDS FROM THE QUERY:**

```
@keywords = $query->keywords
```

If the script was invoked as the result of an <ISINDEX> search, the parsed keywords can be obtained as an array using the *keywords()* method.

**FETCHING THE NAMES OF ALL THE PARAMETERS PASSED TO YOUR SCRIPT:**

```
@names = $query->param
```

If the script was invoked with a parameter list (e.g. "name1=value1&name2=value2&name3=value3"), the *param()* method will return the parameter names as a list. If the script was invoked as an <ISINDEX> script, there will be a single parameter named 'keywords'.

NOTE: As of version 1.5, the array of parameter names returned will be in the same order as they were submitted by the browser. Usually this order is the same as the order in which the parameters are defined in the form (however, this isn't part of the spec, and so isn't guaranteed).

**FETCHING THE VALUE OR VALUES OF A SINGLE NAMED PARAMETER:**

```
@values = $query->param('foo');

          -or-

$value = $query->param('foo');
```

Pass the *param()* method a single argument to fetch the value of the named parameter. If the parameter is multivalued (e.g. from multiple selections in a scrolling list), you can ask to receive an array. Otherwise the method will return a single value.

**SETTING THE** *VALUE* (S) OF A NAMED PARAMETER:

```
$query->param('foo','an','array','of','values');
```

This sets the value for the named parameter 'foo' to an array of values. This is one way to change the value of a field AFTER the script has been invoked once before. (Another way is with the −override parameter accepted by all methods that generate form elements.)

*param()* also recognizes a named parameter style of calling described in more detail later:

```
$query->param(-name=>'foo',-values=>['an','array','of','values']);

                          -or-

$query->param(-name=>'foo',-value=>'the value');
```

**APPENDING ADDITIONAL VALUES TO A NAMED PARAMETER:**

```
$query->append(-name=>'foo',-values=>['yet','more','values']);
```

This adds a value or list of values to the named parameter. The values are appended to the end of the parameter if it already exists. Otherwise the parameter is created. Note that this method only recognizes the named argument calling syntax.

**IMPORTING ALL PARAMETERS INTO A NAMESPACE:**

```
$query->import_names('R');
```

This creates a series of variables in the 'R' namespace. For example, `$R::foo`, `@R:foo`. For keyword lists, a variable `@R::keywords` will appear. If no namespace is given, this method will assume 'Q'. WARNING: don't import anything into 'main'; this is a major security risk!!!!

In older versions, this method was called **import()**. As of version 2.20, this name has been removed completely to avoid conflict with the built-in Perl module **import** operator.

**DELETING A PARAMETER COMPLETELY:**

```
$query->delete('foo');
```

This completely clears a parameter. It sometimes useful for resetting parameters that you don't want passed down between script invocations.

If you are using the function call interface, use "*Delete()*" instead to avoid conflicts with Perl's built-in delete operator.

**DELETING ALL PARAMETERS:**

```
$query->delete_all();
```

This clears the CGI object completely. It might be useful to ensure that all the defaults are taken when you create a fill-out form.

Use *Delete_all()* instead if you are using the function call interface.

**DIRECT ACCESS TO THE PARAMETER LIST:**

```
$q->param_fetch('address')->[1] = '1313 Mockingbird Lane';
unshift @{$q->param_fetch(-name=>'address')},'George Munster';
```

If you need access to the parameter list in a way that isn't covered by the methods above, you can obtain a direct reference to it by calling the **param_fetch()** method with the name of the . This will return an array reference to the named parameters, which you then can manipulate in any way you like.

You can also use a named argument style using the **–name** argument.

**SAVING THE STATE OF THE SCRIPT TO A FILE:**

```
$query->save(FILEHANDLE)
```

This will write the current state of the form to the provided filehandle. You can read it back in by providing a filehandle to the *new()* method. Note that the filehandle can be a file, a pipe, or whatever!

The format of the saved file is:

```
NAME1=VALUE1
NAME1=VALUE1'
NAME2=VALUE2
NAME3=VALUE3
=
```

Both name and value are URL escaped. Multi-valued CGI parameters are represented as repeated names. A session record is delimited by a single = symbol. You can write out multiple records and read them back in with several calls to **new**. You can do this across several sessions by opening the file in append mode, allowing you to create primitive guest books, or to keep a history of users' queries. Here's a short example of creating multiple session records:

```
use CGI;

open (OUT,">>test.out") || die;
$records = 5;
foreach (0..$records) {
    my $q = new CGI;
    $q->param(-name=>'counter',-value=>$_);
    $q->save(OUT);
}
close OUT;

# reopen for reading
open (IN,"test.out") || die;
while (!eof(IN)) {
    my $q = new CGI(IN);
    print $q->param('counter'),"\n";
}
```

The file format used for save/restore is identical to that used by the Whitehead Genome Center's data exchange format "Boulderio", and can be manipulated and even databased using Boulderio utilities. See

http://www.genome.wi.mit.edu/genome_software/other/boulder.html

for further details.

If you wish to use this method from the function-oriented (non-OO) interface, the exported name for this method is **save_parameters()**.

### USING THE FUNCTION−ORIENTED INTERFACE

To use the function-oriented interface, you must specify which CGI.pm routines or sets of routines to import into your script's namespace. There is a small overhead associated with this importation, but it isn't much.

```
use CGI <list of methods>;
```

The listed methods will be imported into the current package; you can call them directly without creating a CGI object first. This example shows how to import the **param()** and **header()** methods, and then use them directly:

```
use CGI 'param','header';
print header('text/plain');
$zipcode = param('zipcode');
```

More frequently, you'll import common sets of functions by referring to the gropus by name. All function sets are preceded with a ":" character as in ":html3" (for tags defined in the HTML 3 standard).

Here is a list of the function sets you can import:

**:cgi**
Import all CGI−handling methods, such as **param()**, **path_info()** and the like.

**:form**
Import all fill-out form generating methods, such as **textfield()**.

**:html2**
Import all methods that generate HTML 2.0 standard elements.

**:html3**
Import all methods that generate HTML 3.0 proposed elements (such as <table>, <super> and <sub>).

**:netscape**
Import all methods that generate Netscape-specific HTML extensions.

**:html**
Import all HTML−generating shortcuts (i.e. 'html2' + 'html3' + 'netscape')...

**:standard**
Import "standard" features, 'html2', 'html3', 'form' and 'cgi'.

**:all** Import all the available methods. For the full list, see the CGI.pm code, where the variable `%TAGS` is defined.

If you import a function name that is not part of CGI.pm, the module will treat it as a new HTML tag and generate the appropriate subroutine. You can then use it like any other HTML tag. This is to provide for the rapidly-evolving HTML "standard." For example, say Microsoft comes out with a new tag called <GRADIENT> (which causes the user's desktop to be flooded with a rotating gradient fill until his machine reboots). You don't need to wait for a new version of CGI.pm to start using it immeidately:

```
use CGI qw/:standard :html3 gradient/;
print gradient({-start=>'red',-end=>'blue'});
```

Note that in the interests of execution speed CGI.pm does **not** use the standard the *Exporter* manpage syntax for specifying load symbols. This may change in the future.

If you import any of the state-maintaining CGI or form-generating methods, a default CGI object will be created and initialized automatically the first time you use any of the methods that require one to be present. This includes **param()**, **textfield()**, **submit()** and the like. (If you need direct access to the CGI object, you can find it in the global variable **$CGI::Q**). By importing CGI.pm methods, you can create visually elegant scripts:

```
use CGI qw/:standard/;
print
    header,
    start_html('Simple Script'),
    h1('Simple Script'),
    start_form,
    "What's your name? ",textfield('name'),p,
    "What's the combination?",
    checkbox_group(-name=>'words',
                   -values=>['eenie','meenie','minie','moe'],
                   -defaults=>['eenie','moe']),p,
    "What's your favorite color?",
    popup_menu(-name=>'color',
               -values=>['red','green','blue','chartreuse']),p,
    submit,
    end_form,
    hr,"\n";

 if (param) {
    print
        "Your name is ",em(param('name')),p,
        "The keywords are: ",em(join(", ",param('words'))),p,
        "Your favorite color is ",em(param('color')),".\n";
 }
 print end_html;
```

**PRAGMAS**

In addition to the function sets, there are a number of pragmas that you can import. Pragmas, which are always preceded by a hyphen, change the way that CGI.pm functions in various ways. Pragmas, function sets, and individual functions can all be imported in the same *use()* line. For example, the following use statement imports the standard set of functions and disables debugging mode (pragma −no_debug):

```
use CGI qw/:standard -no_debug/;
```

The current list of pragmas is as follows:

-any

When you *use CGI −any*, then any method that the query object doesn't recognize will be interpreted as a new HTML tag. This allows you to support the next *ad hoc* Netscape or Microsoft HTML extension. This lets you go wild with new and unsupported tags:

```
use CGI qw(-any);
$q=new CGI;
print $q->gradient({speed=>'fast',start=>'red',end=>'blue'});
```

Since using <cite>any</cite> causes any mistyped method name to be interpreted as an HTML tag, use it with care or not at all.

-compile

This causes the indicated autoloaded methods to be compiled up front, rather than deferred to later. This is useful for scripts that run for an extended period of time under FastCGI or mod_perl, and for those destined to be crunched by Malcom Beattie's Perl compiler. Use it in conjunction with the methods or method familes you plan to use.

```
use CGI qw(-compile :standard :html3);
```

or even

```
use CGI qw(-compile :all);
```

Note that using the −compile pragma in this way will always have the effect of importing the compiled functions into the current namespace. If you want to compile without importing use the *compile()* method instead (see below).

-nph

This makes CGI.pm produce a header appropriate for an NPH (no parsed header) script. You may need to do other things as well to tell the server that the script is NPH. See the discussion of NPH scripts below.

-autoload

This overrides the autoloader so that any function in your program that is not recognized is referred to CGI.pm for possible evaluation. This allows you to use all the CGI.pm functions without adding them to your symbol table, which is of concern for mod_perl users who are worried about memory consumption. *Warning:* when *−autoload* is in effect, you cannot use "poetry mode" (functions without the parenthesis). Use *hr()* rather than *hr*, or add something like *use subs qw/hr p header/* to the top of your script.

-no_debug

This turns off the command-line processing features. If you want to run a CGI.pm script from the command line to produce HTML, and you don't want it pausing to request CGI parameters from standard input or the command line, then use this pragma:

```
use CGI qw(-no_debug :standard);
```

If you'd like to process the command-line parameters but not standard input, this should work:

```
    use CGI qw(-no_debug :standard);
    restore_parameters(join('&',@ARGV));

See the section on debugging for more details.
```

-private_tempfiles
: CGI.pm can process uploaded file. Ordinarily it spools the uploaded file to a temporary directory, then deletes the file when done. However, this opens the risk of eavesdropping as described in the file upload section. Another CGI script author could peek at this data during the upload, even if it is confidential information. On Unix systems, the −private_tempfiles pragma will cause the temporary file to be unlinked as soon as it is opened and before any data is written into it, eliminating the risk of eavesdropping. n =back

## GENERATING DYNAMIC DOCUMENTS

Most of CGI.pm's functions deal with creating documents on the fly. Generally you will produce the HTTP header first, followed by the document itself. CGI.pm provides functions for generating HTTP headers of various types as well as for generating HTML. For creating GIF images, see the GD.pm module.

Each of these functions produces a fragment of HTML or HTTP which you can print out directly so that it displays in the browser window, append to a string, or save to a file for later use.

### CREATING A STANDARD HTTP HEADER:

Normally the first thing you will do in any CGI script is print out an HTTP header. This tells the browser what type of document to expect, and gives other optional information, such as the language, expiration date, and whether to cache the document. The header can also be manipulated for special purposes, such as server push and pay per view pages.

```
print $query->header;

     -or-

print $query->header('image/gif');

     -or-

print $query->header('text/html','204 No response');

     -or-

print $query->header(-type=>'image/gif',
                     -nph=>1,
                     -status=>'402 Payment required',
                     -expires=>'+3d',
                     -cookie=>$cookie,
                     -Cost=>'$2.00');
```

*header()* returns the Content-type: header. You can provide your own MIME type if you choose, otherwise it defaults to text/html. An optional second parameter specifies the status code and a human-readable message. For example, you can specify 204, "No response" to create a script that tells the browser to do nothing at all.

The last example shows the named argument style for passing arguments to the CGI methods using named parameters. Recognized parameters are **−type**, **−status**, **−expires**, and **−cookie**. Any other named parameters will be stripped of their initial hyphens and turned into header fields, allowing you to specify any HTTP header you desire. Internal underscores will be turned into hyphens:

```
print $query->header(-Content_length=>3002);
```

Most browsers will not cache the output from CGI scripts. Every time the browser reloads the page, the script is invoked anew. You can change this behavior with the **−expires** parameter. When you specify an absolute or relative expiration interval with this parameter, some browsers and proxy servers will cache the script's output until the indicated expiration date. The following forms are all valid for the −expires field:

```
+30s                              30 seconds from now
+10m                              ten minutes from now
+1h                               one hour from now
-1d                               yesterday (i.e. "ASAP!")
now                               immediately
+3M                               in three months
+10y                              in ten years time
Thursday, 25-Apr-1999 00:40:33 GMT  at the indicated time & date
```

The **−cookie** parameter generates a header that tells the browser to provide a "magic cookie" during all subsequent transactions with your script. Netscape cookies have a special format that includes interesting attributes such as expiration time. Use the *cookie()* method to create and retrieve session cookies.

The **−nph** parameter, if set to a true value, will issue the correct headers to work with a NPH (no-parse-header) script. This is important to use with certain servers, such as Microsoft Internet Explorer, which expect all their scripts to be NPH.

### GENERATING A REDIRECTION HEADER

```
print $query->redirect('http://somewhere.else/in/movie/land');
```

Sometimes you don't want to produce a document yourself, but simply redirect the browser elsewhere, perhaps choosing a URL based on the time of day or the identity of the user.

The *redirect()* function redirects the browser to a different URL. If you use redirection like this, you should **not** print out a header as well. As of version 2.0, we produce both the unofficial Location: header and the official URI: header. This should satisfy most servers and browsers.

One hint I can offer is that relative links may not work correctly when you generate a redirection to another document on your site. This is due to a well-intentioned optimization that some servers use. The solution to this is to use the full URL (including the http: part) of the document you are redirecting to.

You can also use named arguments:

```
print $query->redirect(-uri=>'http://somewhere.else/in/movie/land',
                       -nph=>1);
```

The **−nph** parameter, if set to a true value, will issue the correct headers to work with a NPH (no-parse-header) script. This is important to use with certain servers, such as Microsoft Internet Explorer, which expect all their scripts to be NPH.

### CREATING THE HTML DOCUMENT HEADER

```
print $query->start_html(-title=>'Secrets of the Pyramids',
                         -author=>'fred@capricorn.org',
                         -base=>'true',
                         -target=>'_blank',
                         -meta=>{'keywords'=>'pharaoh secret mummy',
                                 'copyright'=>'copyright 1996 King Tut'},
                         -style=>{'src'=>'/styles/style1.css'},
                         -BGCOLOR=>'blue');
```

After creating the HTTP header, most CGI scripts will start writing out an HTML document. The *start_html()* routine creates the top of the page, along with a lot of optional information that controls the page's appearance and behavior.

This method returns a canned HTML header and the opening <BODY> tag. All parameters are optional. In the named parameter form, recognized parameters are –title, –author, –base, –xbase and –target (see below for the explanation). Any additional parameters you provide, such as the Netscape unofficial BGCOLOR attribute, are added to the <BODY> tag. Additional parameters must be proceeded by a hyphen.

The argument **–xbase** allows you to provide an HREF for the <BASE> tag different from the current location, as in

```
-xbase=>"http://home.mcom.com/"
```

All relative links will be interpreted relative to this tag.

The argument **–target** allows you to provide a default target frame for all the links and fill-out forms on the page. See the Netscape documentation on frames for details of how to manipulate this.

```
-target=>"answer_window"
```

All relative links will be interpreted relative to this tag. You add arbitrary meta information to the header with the **–meta** argument. This argument expects a reference to an associative array containing name/value pairs of meta information. These will be turned into a series of header <META> tags that look something like this:

```
<META NAME="keywords" CONTENT="pharaoh secret mummy">
<META NAME="description" CONTENT="copyright 1996 King Tut">
```

There is no support for the HTTP–EQUIV type of <META> tag. This is because you can modify the HTTP header directly with the **header()** method. For example, if you want to send the Refresh: header, do it in the *header()* method:

```
print $q->header(-Refresh=>'10; URL=http://www.capricorn.com');
```

The **–style** tag is used to incorporate cascading stylesheets into your code. See the section on CASCADING STYLESHEETS for more information.

You can place other arbitrary HTML elements to the <HEAD> section with the **–head** tag. For example, to place the rarely-used <LINK> element in the head section, use this:

```
print $q->start_html(-head=>Link({-rel=>'next',
                                  -href=>'http://www.capricorn.com/s2.html'}));
```

To incorporate multiple HTML elements into the <HEAD> section, just pass an array reference:

```
print $q->start_html(-head=>[
                          Link({-rel=>'next',
                                -href=>'http://www.capricorn.com/s2.html'}),
                          Link({-rel=>'previous',
                                -href=>'http://www.capricorn.com/s1.html'})
                         ]
                 );
```

JAVASCRIPTING: The **–script**, **–noScript**, **–onLoad**, **–onMouseOver**, **–onMouseOut** and **–onUnload** parameters are used to add Netscape JavaScript calls to your pages. **–script** should point to a block of text containing JavaScript function definitions. This block will be placed within a <SCRIPT> block inside the HTML (not HTTP) header. The block is placed in the header in order to give your page a fighting chance of having all its JavaScript functions in place even if the user presses the stop button before the page has loaded completely. CGI.pm attempts to format the script in such a way that JavaScript-naive browsers will not choke on the code: unfortunately there are some browsers, such as Chimera for Unix, that get confused by it nevertheless.

The **–onLoad** and **–onUnload** parameters point to fragments of JavaScript code to execute when the page is respectively opened and closed by the browser. Usually these parameters are calls to functions defined in the **–script** field:

```
$query = new CGI;
print $query->header;
$JSCRIPT=<<END;
// Ask a silly question
function riddle_me_this() {
   var r = prompt("What walks on four legs in the morning, " +
                  "two legs in the afternoon, " +
                  "and three legs in the evening?");
   response(r);
}
// Get a silly answer
function response(answer) {
   if (answer == "man")
      alert("Right you are!");
   else
      alert("Wrong!  Guess again.");
}
END
print $query->start_html(-title=>'The Riddle of the Sphinx',
                         -script=>$JSCRIPT);
```

Use the **–noScript** parameter to pass some HTML text that will be displayed on browsers that do not have JavaScript (or browsers where JavaScript is turned off).

Netscape 3.0 recognizes several attributes of the <SCRIPT> tag, including LANGUAGE and SRC. The latter is particularly interesting, as it allows you to keep the JavaScript code in a file or CGI script rather than cluttering up each page with the source. To use these attributes pass a HASH reference in the **–script** parameter containing one or more of –language, –src, or –code:

```
print $q->start_html(-title=>'The Riddle of the Sphinx',
                     -script=>{-language=>'JAVASCRIPT',
                               -src=>'/javascript/sphinx.js'}
                     );
```

```
print $q->(-title=>'The Riddle of the Sphinx',
           -script=>{-language=>'PERLSCRIPT'},
                     -code=>'print "hello world!\n;"'
           );
```

A final feature allows you to incorporate multiple <SCRIPT> sections into the header. Just pass the list of script sections as an array reference. this allows you to specify different source files for different dialects of JavaScript. Example:

```
print $q-&gt;start_html(-title=&gt;'The Riddle of the Sphinx',
                     -script=&gt;[
                               { -language =&gt; 'JavaScript1.0',
                                 -src      =&gt; '/javascript/utilities10.js'
                               },
                               { -language =&gt; 'JavaScript1.1',
                                 -src      =&gt; '/javascript/utilities11.js'
                               },
                               { -language =&gt; 'JavaScript1.2',
                                 -src      =&gt; '/javascript/utilities12.js'
                               },
                               { -language =&gt; 'JavaScript28.2',
                                 -src      =&gt; '/javascript/utilities219.js'
                               }
                            ]
                        );
</pre>
```

If this looks a bit extreme, take my advice and stick with straight CGI scripting.

See

```
http://home.netscape.com/eng/mozilla/2.0/handbook/javascript/
```

for more information about JavaScript.

The old-style positional parameters are as follows:

**Parameters:**

1. The title
2. The author's e-mail address (will create a <LINK REV="MADE"> tag if present
3. A 'true' flag if you want to include a <BASE> tag in the header. This helps resolve relative addresses to absolute ones when the document is moved, but makes the document hierarchy non-portable. Use with care!

4, 5, 6... Any other parameters you want to include in the <BODY> tag. This is a good place to put Netscape extensions, such as colors and wallpaper patterns.

**ENDING THE HTML DOCUMENT:**

```
print $query->end_html
```

This ends an HTML document by printing the </BODY></HTML> tags.

**CREATING A SELF−REFERENCING URL THAT PRESERVES STATE INFORMATION:**

```
$myself = $query->self_url;
print "<A HREF=$myself>I'm talking to myself.</A>";
```

*self_url()* will return a URL, that, when selected, will reinvoke this script with all its state information intact. This is most useful when you want to jump around within the document using internal anchors but you don't want to disrupt the current contents of the *form* (s). Something like this will do the trick.

```
$myself = $query->self_url;
print "<A HREF=$myself#table1>See table 1</A>";
print "<A HREF=$myself#table2>See table 2</A>";
print "<A HREF=$myself#yourself>See for yourself</A>";
```

If you want more control over what's returned, using the **url()** method instead.

You can also retrieve the unprocessed query string with *query_string()*:

```
$the_string = $query->query_string;
```

**OBTAINING THE SCRIPT'S URL**

```
$full_url      = $query->url();
$full_url      = $query->url(-full=>1);  #alternative syntax
$relative_url  = $query->url(-relative=>1);
$absolute_url  = $query->url(-absolute=>1);
$url_with_path = $query->url(-path_info=>1);
$url_with_path_and_query = $query->url(-path_info=>1,-query=>1);
```

**url()** returns the script's URL in a variety of formats. Called without any arguments, it returns the full form of the URL, including host name and port number

```
http://your.host.com/path/to/script.cgi
```

You can modify this format with the following named arguments:

**−absolute**
If true, produce an absolute URL, e.g.

```
/path/to/script.cgi
```

**−relative**
Produce a relative URL. This is useful if you want to reinvoke your script with different parameters. For example:

```
script.cgi
```

**−full** Produce the full URL, exactly as if called without any arguments. This overrides the −relative and −absolute arguments.

**−path** (**−path_info**)
Append the additional path information to the URL. This can be combined with **−full**, **−absolute** or **−relative**. **−path_info** is provided as a synonym.

**−query** (**−query_string**)
Append the query string to the URL. This can be combined with **−full**, **−absolute** or **−relative**. **−query_string** is provided as a synonym.

**CREATING STANDARD HTML ELEMENTS:**

CGI.pm defines general HTML shortcut methods for most, if not all of the HTML 3 and HTML 4 tags. HTML shortcuts are named after a single HTML element and return a fragment of HTML text that you can then print or manipulate as you like. Each shortcut returns a fragment of HTML code that you can append to a string, save to a file, or, most commonly, print out so that it displays in the browser window.

This example shows how to use the HTML methods:

```
$q = new CGI;
print $q->blockquote(
                  "Many years ago on the island of",
                  $q->a({href=>"http://crete.org/"},"Crete"),
                  "there lived a minotaur named",
                  $q->strong("Fred."),
                 ),
    $q->hr;
```

This results in the following HTML code (extra newlines have been added for readability):

```
<blockquote>
Many years ago on the island of
<a HREF="http://crete.org/">Crete</a> there lived
a minotaur named <strong>Fred.</strong>
</blockquote>
<hr>
```

If you find the syntax for calling the HTML shortcuts awkward, you can import them into your namespace and dispense with the object syntax completely (see the next section for more details):

```
use CGI ':standard';
print blockquote(
   "Many years ago on the island of",
   a({href=>"http://crete.org/"},"Crete"),
   "there lived a minotaur named",
   strong("Fred."),
   ),
   hr;
```

**PROVIDING ARGUMENTS TO HTML SHORTCUTS**

The HTML methods will accept zero, one or multiple arguments. If you provide no arguments, you get a single tag:

```
print hr;    #  <HR>
```

If you provide one or more string arguments, they are concatenated together with spaces and placed between opening and closing tags:

```
print h1("Chapter","1"); # <H1>Chapter 1</H1>"
```

If the first argument is an associative array reference, then the keys and values of the associative array become the HTML tag's attributes:

```
print a({-href=>'fred.html',-target=>'_new'},
   "Open a new frame");
```

```
        <A HREF="fred.html",TARGET="_new">Open a new frame</A>
```

```
You may dispense with the dashes in front of the attribute names if
you prefer:

   print img {src=>'fred.gif',align=>'LEFT'};

           <IMG ALIGN="LEFT" SRC="fred.gif">
```

Sometimes an HTML tag attribute has no argument. For example, ordered lists can be marked as COMPACT. The syntax for this is an argument that that points to an undef string:

```
print ol({compact=>undef},li('one'),li('two'),li('three'));
```

Prior to CGI.pm version 2.41, providing an empty ('') string as an attribute argument was the same as providing undef. However, this has changed in order to accomodate those who want to create tags of the form <IMG ALT="">. The difference is shown in these two pieces of code:

| CODE | RESULT |
|---|---|
| *img*({alt=>undef}) | <IMG ALT> |
| *img*({alt=>''}) | <IMT ALT=""> |

**THE DISTRIBUTIVE PROPERTY OF HTML SHORTCUTS**

One of the cool features of the HTML shortcuts is that they are distributive. If you give them an argument consisting of a **reference** to a list, the tag will be distributed across each element of the list. For example, here's one way to make an ordered list:

```
print ul(
          li({-type=>'disc'},['Sneezy','Doc','Sleepy','Happy']);
        );
```

This example will result in HTML output that looks like this:

```
<UL>
  <LI TYPE="disc">Sneezy</LI>
  <LI TYPE="disc">Doc</LI>
  <LI TYPE="disc">Sleepy</LI>
  <LI TYPE="disc">Happy</LI>
</UL>
```

This is extremely useful for creating tables. For example:

```
print table({-border=>undef},
        caption('When Should You Eat Your Vegetables?'),
        Tr({-align=>CENTER,-valign=>TOP},
        [
           th(['Vegetable', 'Breakfast','Lunch','Dinner']),
           td(['Tomatoes' , 'no', 'yes', 'yes']),
           td(['Broccoli' , 'no', 'no',  'yes']),
           td(['Onions'   , 'yes','yes', 'yes'])
        ]
        )
     );
```

### HTML SHORTCUTS AND LIST INTERPOLATION

Consider this bit of code:

```
print blockquote(em('Hi'),'mom!'));
```

It will ordinarily return the string that you probably expect, namely:

```
<BLOCKQUOTE><EM>Hi</EM> mom!</BLOCKQUOTE>
```

Note the space between the element "Hi" and the element "mom!". CGI.pm puts the extra space there using array interpolation, which is controlled by the magic $" variable. Sometimes this extra space is not what you want, for example, when you are trying to align a series of images. In this case, you can simply change the value of $" to an empty string.

```
{
   local($") = '';
   print blockquote(em('Hi'),'mom!'));
 }
```

I suggest you put the code in a block as shown here. Otherwise the change to $" will affect all subsequent code until you explicitly reset it.

### NON−STANDARD HTML SHORTCUTS

A few HTML tags don't follow the standard pattern for various reasons.

**comment()** generates an HTML comment (<!-- comment -->). Call it like

```
print comment('here is my comment');
```

Because of conflicts with built-in Perl functions, the following functions begin with initial caps:

```
Select
Tr
Link
Delete
```

In addition, *start_html()*, *end_html()*, *start_form()*, *end_form()*, *start_multipart_form()* and all the fill-out form tags are special. See their respective sections.

## CREATING FILL−OUT FORMS:

*General note* The various form-creating methods all return strings to the caller, containing the tag or tags that will create the requested form element. You are responsible for actually printing out these strings. It's set up this way so that you can place formatting tags around the form elements.

*Another note* The default values that you specify for the forms are only used the **first** time the script is invoked (when there is no query string). On subsequent invocations of the script (when there is a query string), the former values are used even if they are blank.

If you want to change the value of a field from its previous value, you have two choices:

(1) call the *param()* method to set it.

(2) use the −override (alias −force) parameter (a new feature in version 2.15). This forces the default value to be used, regardless of the previous value:

```
print $query->textfield(-name=>'field_name',
                        -default=>'starting value',
                        -override=>1,
                        -size=>50,
                        -maxlength=>80);
```

*Yet another note* By default, the text and labels of form elements are escaped according to HTML rules. This means that you can safely use "<CLICK ME>" as the label for a button. However, it also interferes with your ability to incorporate special HTML character sequences, such as Á, into your fields. If you wish to turn off automatic escaping, call the *autoEscape()* method with a false value immediately after creating the CGI object:

```
$query = new CGI;
$query->autoEscape(undef);
```

**CREATING AN ISINDEX TAG**

```
print $query->isindex(-action=>$action);

      -or-

print $query->isindex($action);
```

Prints out an <ISINDEX> tag. Not very exciting. The parameter –action specifies the URL of the script to process the query. The default is to process the query with the current script.

**STARTING AND ENDING A FORM**

```
print $query->startform(-method=>$method,
                        -action=>$action,
                        -encoding=>$encoding);
  <... various form stuff ...>
print $query->endform;

    -or-

print $query->startform($method,$action,$encoding);
  <... various form stuff ...>
print $query->endform;
```

*startform()* will return a <FORM> tag with the optional method, action and form encoding that you specify. The defaults are:
 method: POST
 action: this script
 encoding: application/x-www-form-urlencoded

*endform()* returns the closing </FORM> tag.

*Startform()*'s encoding method tells the browser how to package the various fields of the form before sending the form to the server. Two values are possible:

**application/x-www-form-urlencoded**
This is the older type of encoding used by all browsers prior to Netscape 2.0. It is compatible with many CGI scripts and is suitable for short fields containing text data. For your convenience, CGI.pm stores the name of this encoding type in **$CGI::URL_ENCODED**.

**multipart/form-data**

This is the newer type of encoding introduced by Netscape 2.0. It is suitable for forms that contain very large fields or that are intended for transferring binary data. Most importantly, it enables the "file upload" feature of Netscape 2.0 forms. For your convenience, CGI.pm stores the name of this encoding type in **&CGI::MULTIPART**

Forms that use this type of encoding are not easily interpreted by CGI scripts unless they use CGI.pm or another library designed to handle them.

For compatibility, the *startform()* method uses the older form of encoding by default. If you want to use the newer form of encoding by default, you can call **start_multipart_form()** instead of **startform()**.

JAVASCRIPTING: The **−name** and **−onSubmit** parameters are provided for use with JavaScript. The −name parameter gives the form a name so that it can be identified and manipulated by JavaScript functions. −onSubmit should point to a JavaScript function that will be executed just before the form is submitted to your server. You can use this opportunity to check the contents of the form for consistency and completeness. If you find something wrong, you can put up an alert box or maybe fix things up yourself. You can abort the submission by returning false from this function.

Usually the bulk of JavaScript functions are defined in a <SCRIPT> block in the HTML header and −onSubmit points to one of these function call. See *start_html()* for details.

**CREATING A TEXT FIELD**

```
print $query->textfield(-name=>'field_name',
                        -default=>'starting value',
                        -size=>50,
                        -maxlength=>80);
    -or-

print $query->textfield('field_name','starting value',50,80);
```

*textfield()* will return a text input field.

**Parameters**

1. The first parameter is the required name for the field (−name).
2. The optional second parameter is the default starting value for the field contents (−default).
3. The optional third parameter is the size of the field in
   characters (−size).
4. The optional fourth parameter is the maximum number of characters the
   field will accept (−maxlength).

As with all these methods, the field will be initialized with its previous contents from earlier invocations of the script. When the form is processed, the value of the text field can be retrieved with:

```
$value = $query->param('foo');
```

If you want to reset it from its initial value after the script has been called once, you can do so like this:

```
$query->param('foo',"I'm taking over this value!");
```

NEW AS OF VERSION 2.15: If you don't want the field to take on its previous value, you can force its current value by using the −override (alias −force) parameter:

```
print $query->textfield(-name=>'field_name',
                        -default=>'starting value',
                        -override=>1,
                        -size=>50,
                        -maxlength=>80);
```

JAVASCRIPTING: You can also provide **−onChange**, **−onFocus**, **−onBlur**, **−onMouseOver**, **−onMouseOut** and **−onSelect** parameters to register JavaScript event handlers. The onChange handler will be called whenever the user changes the contents of the text field. You can do text validation if you like. onFocus and onBlur are called respectively when the insertion point moves into and out of the text field. onSelect is called when the user changes the portion of the text that is selected.

**CREATING A BIG TEXT FIELD**

```
print $query->textarea(-name=>'foo',
                       -default=>'starting value',
                       -rows=>10,
                       -columns=>50);

     -or

print $query->textarea('foo','starting value',10,50);
```

*textarea()* is just like textfield, but it allows you to specify rows and columns for a multiline text entry box. You can provide a starting value for the field, which can be long and contain multiple lines.

JAVASCRIPTING: The **−onChange**, **−onFocus**, **−onBlur** , **−onMouseOver**, **−onMouseOut**, and **−onSelect** parameters are recognized. See *textfield()*.

**CREATING A PASSWORD FIELD**

```
print $query->password_field(-name=>'secret',
                             -value=>'starting value',
                             -size=>50,
                             -maxlength=>80);
     -or-

print $query->password_field('secret','starting value',50,80);
```

*password_field()* is identical to *textfield()*, except that its contents will be starred out on the web page.

JAVASCRIPTING: The **−onChange**, **−onFocus**, **−onBlur**, **−onMouseOver**, **−onMouseOut** and **−onSelect** parameters are recognized. See *textfield()*.

**CREATING A FILE UPLOAD FIELD**

```
 print $query->filefield(-name=>'uploaded_file',
                         -default=>'starting value',
                         -size=>50,
                         -maxlength=>80);
     -or-

 print $query->filefield('uploaded_file','starting value',50,80);
```

*filefield()* will return a file upload field for Netscape 2.0 browsers. In order to take full advantage of this *you must use the new multipart encoding scheme* for the form. You can do this either by calling **startform()** with an encoding type of **$CGI::MULTIPART**, or by calling the new method **start_multipart_form()** instead of vanilla **startform()**.

**Parameters**

1. The first parameter is the required name for the field (−name).
2. The optional second parameter is the starting value for the field contents to be used as the default file name (−default).

   The beta2 version of Netscape 2.0 currently doesn't pay any attention to this field, and so the starting value will always be blank. Worse, the field loses its "sticky" behavior and forgets its previous contents. The starting value field is called for in the HTML specification, however, and possibly later versions of Netscape will honor it.
3. The optional third parameter is the size of the field in characters (−size).
4. The optional fourth parameter is the maximum number of characters the field will accept (−maxlength).

   When the form is processed, you can retrieve the entered filename by calling *param()*.

   ```
   $filename = $query->param('uploaded_file');
   ```

   In Netscape Navigator 2.0, the filename that gets returned is the full local filename on the **remote user's** machine. If the remote user is on a Unix machine, the filename will follow Unix conventions:

   ```
   /path/to/the/file
   ```

   On an MS−DOS/Windows and OS/2 machines, the filename will follow DOS conventions:

   ```
   C:\PATH\TO\THE\FILE.MSW
   ```

   On a Macintosh machine, the filename will follow Mac conventions:

   ```
   HD 40:Desktop Folder:Sort Through:Reminders
   ```

   The filename returned is also a file handle. You can read the contents of the file using standard Perl file reading calls:

   ```
   # Read a text file and print it out
   while (<$filename>) {
      print;
   }

   # Copy a binary file to somewhere safe
   open (OUTFILE,">>/usr/local/web/users/feedback");
   while ($bytesread=read($filename,$buffer,1024)) {
      print OUTFILE $buffer;
   }
   ```

   When a file is uploaded the browser usually sends along some information along with it in the format of headers. The information usually includes the MIME content type. Future browsers may send other information as well (such as modification date and size). To retrieve this information, call *uploadInfo()*. It returns a reference to an associative array containing all the document headers.

```
$filename = $query->param('uploaded_file');
$type = $query->uploadInfo($filename)->{'Content-Type'};
unless ($type eq 'text/html') {
   die "HTML FILES ONLY!";
}
```

If you are using a machine that recognizes "text" and "binary" data modes, be sure to understand when and how to use them (see the Camel book). Otherwise you may find that binary files are corrupted during file uploads.

JAVASCRIPTING: The **–onChange**, **–onFocus**, **–onBlur**, **–onMouseOver**, **–onMouseOut** and **–onSelect** parameters are recognized. See *textfield()* for details.

**CREATING A POPUP MENU**

```
print $query->popup_menu('menu_name',
                         ['eenie','meenie','minie'],
                         'meenie');

   -or-

%labels = ('eenie'=>'your first choice',
           'meenie'=>'your second choice',
           'minie'=>'your third choice');
print $query->popup_menu('menu_name',
                         ['eenie','meenie','minie'],
                         'meenie',\%labels);

     -or (named parameter style)-

print $query->popup_menu(-name=>'menu_name',
                         -values=>['eenie','meenie','minie'],
                         -default=>'meenie',
                         -labels=>\%labels);
```

*popup_menu()* creates a menu.

1. The required first argument is the menu's name (–name).
2. The required second argument (–values) is an array **reference** containing the list of menu items in the menu. You can pass the method an anonymous array, as shown in the example, or a reference to a named array, such as "\@foo".
3. The optional third parameter (–default) is the name of the default menu choice. If not specified, the first item will be the default. The values of the previous choice will be maintained across queries.
4. The optional fourth parameter (–labels) is provided for people who want to use different values for the user-visible label inside the popup menu nd the value returned to your script. It's a pointer to an associative array relating menu values to user-visible labels. If you leave this parameter blank, the menu values will be displayed by default. (You can also leave a label undefined if you want to).

   When the form is processed, the selected value of the popup menu can be retrieved using:

   ```
   $popup_menu_value = $query->param('menu_name');
   ```

   JAVASCRIPTING: *popup_menu()* recognizes the following event handlers: **–onChange**, **–onFocus**, **–onMouseOver**, **–onMouseOut**, and **–onBlur**. See the *textfield()* section for details on when these handlers are called.

### CREATING A SCROLLING LIST

```
print $query->scrolling_list('list_name',
                             ['eenie','meenie','minie','moe'],
                             ['eenie','moe'],5,'true');
   -or-

print $query->scrolling_list('list_name',
                             ['eenie','meenie','minie','moe'],
                             ['eenie','moe'],5,'true',
                             \%labels);

     -or-

print $query->scrolling_list(-name=>'list_name',
                             -values=>['eenie','meenie','minie','moe'],
                             -default=>['eenie','moe'],
                             -size=>5,
                             -multiple=>'true',
                             -labels=>\%labels);
```

*scrolling_list()* creates a scrolling list.

**Parameters:**

1. The first and second arguments are the list name (–name) and values (–values). As in the popup menu, the second argument should be an array reference.
2. The optional third argument (–default) can be either a reference to a list containing the values to be selected by default, or can be a single value to select. If this argument is missing or undefined, then nothing is selected when the list first appears. In the named parameter version, you can use the synonym "–defaults" for this parameter.
3. The optional fourth argument is the size of the list (–size).
4. The optional fifth argument can be set to true to allow multiple simultaneous selections (–multiple). Otherwise only one selection will be allowed at a time.
5. The optional sixth argument is a pointer to an associative array containing long user-visible labels for the list items (–labels). If not provided, the values will be displayed.

   When this form is processed, all selected list items will be returned as a list under the parameter name 'list_name'. The values of the selected items can be retrieved with:

   ```
   @selected = $query->param('list_name');
   ```

   JAVASCRIPTING: *scrolling_list()* recognizes the following event handlers: **–onChange**, **–onFocus**, **–onMouseOver**, **–onMouseOut** and **–onBlur**. See *textfield()* for the description of when these handlers are called.

### CREATING A GROUP OF RELATED CHECKBOXES

```
print $query->checkbox_group(-name=>'group_name',
                             -values=>['eenie','meenie','minie','moe'],
                             -default=>['eenie','moe'],
                             -linebreak=>'true',
                             -labels=>\%labels);

print $query->checkbox_group('group_name',
                             ['eenie','meenie','minie','moe'],
                             ['eenie','moe'],'true',\%labels);

HTML3-COMPATIBLE BROWSERS ONLY:

print $query->checkbox_group(-name=>'group_name',
                             -values=>['eenie','meenie','minie','moe'],
                             -rows=2,-columns=>2);
```

*checkbox_group()* creates a list of checkboxes that are related by the same name.

**Parameters:**

1. The first and second arguments are the checkbox name and values, respectively (–name and –values). As in the popup menu, the second argument should be an array reference. These values are used for the user-readable labels printed next to the checkboxes as well as for the values passed to your script in the query string.
2. The optional third argument (–default) can be either a reference to a list containing the values to be checked by default, or can be a single value to checked. If this argument is missing or undefined, then nothing is selected when the list first appears.
3. The optional fourth argument (–linebreak) can be set to true to place line breaks between the checkboxes so that they appear as a vertical list. Otherwise, they will be strung together on a horizontal line.
4. The optional fifth argument is a pointer to an associative array relating the checkbox values to the user-visible labels that will be printed next to them (–labels). If not provided, the values will be used as the default.
5. **HTML3-compatible browsers** (such as Netscape) can take advantage of the optional parameters **–rows**, and **–columns**. These parameters cause *checkbox_group()* to return an HTML3 compatible table containing the checkbox group formatted with the specified number of rows and columns. You can provide just the –columns parameter if you wish; checkbox_group will calculate the correct number of rows for you.

   To include row and column headings in the returned table, you can use the **–rowheaders** and **–colheaders** parameters. Both of these accept a pointer to an array of headings to use. The headings are just decorative. They don't reorganize the interpretation of the checkboxes -- they're still a single named unit.

   When the form is processed, all checked boxes will be returned as a list under the parameter name 'group_name'. The values of the "on" checkboxes can be retrieved with:

   ```
   @turned_on = $query->param('group_name');
   ```

   The value returned by *checkbox_group()* is actually an array of button elements. You can capture them and use them within tables, lists, or in other creative ways:

   ```
   @h = $query->checkbox_group(-name=>'group_name',-values=>\@values);
   &use_in_creative_way(@h);
   ```

JAVASCRIPTING: *checkbox_group()* recognizes the **−onClick** parameter. This specifies a JavaScript code fragment or function call to be executed every time the user clicks on any of the buttons in the group. You can retrieve the identity of the particular button clicked on using the "this" variable.

**CREATING A STANDALONE CHECKBOX**

```
    print $query->checkbox(-name=>'checkbox_name',
                           -checked=>'checked',
                           -value=>'ON',
                           -label=>'CLICK ME');

        -or-

    print $query->checkbox('checkbox_name','checked','ON','CLICK ME');
```

*checkbox()* is used to create an isolated checkbox that isn't logically related to any others.

**Parameters:**

1. The first parameter is the required name for the checkbox (−name). It will also be used for the user-readable label printed next to the checkbox.
2. The optional second parameter (−checked) specifies that the checkbox is turned on by default. Synonyms are −selected and −on.
3. The optional third parameter (−value) specifies the value of the checkbox when it is checked. If not provided, the word "on" is assumed.
4. The optional fourth parameter (−label) is the user-readable label to be attached to the checkbox. If not provided, the checkbox name is used.

   The value of the checkbox can be retrieved using:

```
       $turned_on = $query->param('checkbox_name');
```

   JAVASCRIPTING: *checkbox()* recognizes the **−onClick** parameter. See *checkbox_group()* for further details.

**CREATING A RADIO BUTTON GROUP**

```
   print $query->radio_group(-name=>'group_name',
                             -values=>['eenie','meenie','minie'],
                             -default=>'meenie',
                             -linebreak=>'true',
                             -labels=>\%labels);

        -or-

   print $query->radio_group('group_name',['eenie','meenie','minie'],
                                          'meenie','true',\%labels);

   HTML3-COMPATIBLE BROWSERS ONLY:

   print $query->radio_group(-name=>'group_name',
                             -values=>['eenie','meenie','minie','moe'],
                             -rows=2,-columns=>2);
```

*radio_group()* creates a set of logically-related radio buttons (turning one member of the group on turns the others off)

**Parameters:**

1. The first argument is the name of the group and is required (−name).
2. The second argument (−values) is the list of values for the radio buttons. The values and the labels that appear on the page are identical. Pass an array *reference* in the second argument, either using an anonymous array, as shown, or by referencing a named array as in "\@foo".
3. The optional third parameter (−default) is the name of the default button to turn on. If not specified, the first item will be the default. You can provide a nonexistent button name, such as "−" to start up with no buttons selected.
4. The optional fourth parameter (−linebreak) can be set to 'true' to put line breaks between the buttons, creating a vertical list.
5. The optional fifth parameter (−labels) is a pointer to an associative array relating the radio button values to user-visible labels to be used in the display. If not provided, the values themselves are displayed.
6. **HTML3-compatible browsers** (such as Netscape) can take advantage of the optional parameters **−rows**, and **−columns**. These parameters cause *radio_group()* to return an HTML3 compatible table containing the radio group formatted with the specified number of rows and columns. You can provide just the −columns parameter if you wish; radio_group will calculate the correct number of rows for you.

   To include row and column headings in the returned table, you can use the **−rowheader** and **−colheader** parameters. Both of these accept a pointer to an array of headings to use. The headings are just decorative. They don't reorganize the interpetation of the radio buttons -- they're still a single named unit.

   When the form is processed, the selected radio button can be retrieved using:

   ```
   $which_radio_button = $query->param('group_name');
   ```

   The value returned by *radio_group()* is actually an array of button elements. You can capture them and use them within tables, lists, or in other creative ways:

   ```
   @h = $query->radio_group(-name=>'group_name',-values=>\@values);
   &use_in_creative_way(@h);
   ```

### CREATING A SUBMIT BUTTON

```
print $query->submit(-name=>'button_name',
                     -value=>'value');

     -or-

print $query->submit('button_name','value');
```

*submit()* will create the query submission button. Every form should have one of these.

**Parameters:**

1. The first argument (−name) is optional. You can give the button a name if you have several submission buttons in your form and you want to distinguish between them. The name will also be used as the user-visible label. Be aware that a few older browsers don't deal with this correctly and **never** send back a value from a button.

2. The second argument (−value) is also optional. This gives the button a value that will be passed to your script in the query string.

   You can figure out which button was pressed by using different values for each one:

```
$which_one = $query->param('button_name');
```

   JAVASCRIPTING: *radio_group()* recognizes the **−onClick** parameter. See *checkbox_group()* for further details.

**CREATING A RESET BUTTON**

```
print $query->reset
```

*reset()* creates the "reset" button. Note that it restores the form to its value from the last time the script was called, NOT necessarily to the defaults.

**CREATING A DEFAULT BUTTON**

```
print $query->defaults('button_label')
```

*defaults()* creates a button that, when invoked, will cause the form to be completely reset to its defaults, wiping out all the changes the user ever made.

**CREATING A HIDDEN FIELD**

```
print $query->hidden(-name=>'hidden_name',
                     -default=>['value1','value2'...]);

        -or-

print $query->hidden('hidden_name','value1','value2'...);
```

*hidden()* produces a text field that can't be seen by the user. It is useful for passing state variable information from one invocation of the script to the next.

**Parameters:**

1. The first argument is required and specifies the name of this field (−name).
2. The second argument is also required and specifies its value (−default). In the named parameter style of calling, you can provide a single value here or a reference to a whole list

   Fetch the value of a hidden field this way:

```
$hidden_value = $query->param('hidden_name');
```

   Note, that just like all the other form elements, the value of a hidden field is "sticky". If you want to replace a hidden field with some other values after the script has been called once you'll have to do it manually:

```
$query->param('hidden_name','new','values','here');
```

**CREATING A CLICKABLE IMAGE BUTTON**

```
print $query->image_button(-name=>'button_name',
                           -src=>'/source/URL',
                           -align=>'MIDDLE');
```

```
   -or-

print $query->image_button('button_name','/source/URL','MIDDLE');
```

*image_button()* produces a clickable image. When it's clicked on the position of the click is returned to your script as "button_name.x" and "button_name.y", where "button_name" is the name you've assigned to it.

JAVASCRIPTING: *image_button()* recognizes the **–onClick** parameter. See *checkbox_group()* for further details.

**Parameters:**

1. The first argument (–name) is required and specifies the name of this field.
2. The second argument (–src) is also required and specifies the URL
3. The third option (–align, optional) is an alignment type, and may be TOP, BOTTOM or MIDDLE

Fetch the value of the button this way:
`$x` = `$query`–>*param* ('button_name.x');
`$y` = `$query`–>*param* ('button_name.y');

**CREATING A JAVASCRIPT ACTION BUTTON**

```
print $query->button(-name=>'button_name',
                     -value=>'user visible label',
                     -onClick=>"do_something()");

   -or-

print $query->button('button_name',"do_something()");
```

*button()* produces a button that is compatible with Netscape 2.0's JavaScript. When it's pressed the fragment of JavaScript code pointed to by the **–onClick** parameter will be executed. On non-Netscape browsers this form element will probably not even display.

**NETSCAPE COOKIES**

Netscape browsers versions 1.1 and higher support a so-called "cookie" designed to help maintain state within a browser session. CGI.pm has several methods that support cookies.

A cookie is a name=value pair much like the named parameters in a CGI query string. CGI scripts create one or more cookies and send them to the browser in the HTTP header. The browser maintains a list of cookies that belong to a particular Web server, and returns them to the CGI script during subsequent interactions.

In addition to the required name=value pair, each cookie has several optional attributes:

1. an expiration time

This is a time/date string (in a special GMT format) that indicates when a cookie expires. The cookie will be saved and returned to your script until this expiration date is reached if the user exits Netscape and restarts it. If an expiration date isn't specified, the cookie will remain active until the user quits Netscape.

2. a domain

This is a partial or complete domain name for which the cookie is valid. The browser will return the cookie to any host that matches the partial domain name. For example, if you specify a domain name of ".capricorn.com", then Netscape will return the cookie to Web servers running on any of the machines "www.capricorn.com", "www2.capricorn.com", "feckless.capricorn.com", etc. Domain names must contain at least two periods to prevent attempts to match on top level domains like ".edu". If no domain is specified, then the browser will only return the cookie to servers on the host the cookie originated from.

3. a path If you provide a cookie path attribute, the browser will check it against your script's URL before returning the cookie. For example, if you specify the path "/cgi-bin", then the cookie will be returned to each of the scripts "/cgi-bin/tally.pl", "/cgi-bin/order.pl", and "/cgi-bin/customer_service/complain.pl", but not to the script "/cgi-private/site_admin.pl". By default, path is set to "/", which causes the cookie to be sent to any CGI script on your site.

4. a "secure" flag
If the "secure" attribute is set, the cookie will only be sent to your script if the CGI request is occurring on a secure channel, such as SSL.

The interface to Netscape cookies is the **cookie()** method:

```
$cookie = $query->cookie(-name=>'sessionID',
                         -value=>'xyzzy',
                         -expires=>'+1h',
                         -path=>'/cgi-bin/database',
                         -domain=>'.capricorn.org',
                         -secure=>1);
print $query->header(-cookie=>$cookie);
```

**cookie()** creates a new cookie. Its parameters include:

**−name** The name of the cookie (required). This can be any string at all. Although Netscape limits its cookie names to non-whitespace alphanumeric characters, CGI.pm removes this restriction by escaping and unescaping cookies behind the scenes.

**−value** The value of the cookie. This can be any scalar value, array reference, or even associative array reference. For example, you can store an entire associative array into a cookie this way:

```
$cookie=$query->cookie(-name=>'family information',
                       -value=>\%childrens_ages);
```

**−path** The optional partial path for which this cookie will be valid, as described above.

**−domain**
The optional partial domain for which this cookie will be valid, as described above.

**−expires**
The optional expiration date for this cookie. The format is as described in the section on the **header()** method:

```
"+1h"  one hour from now
```

**−secure** If set to true, this cookie will only be used within a secure SSL session.

The cookie created by *cookie()* must be incorporated into the HTTP header within the string returned by the *header()* method:

```
print $query->header(-cookie=>$my_cookie);
```

To create multiple cookies, give *header()* an array reference:

```
$cookie1 = $query->cookie(-name=>'riddle_name',
                          -value=>"The Sphynx's Question");
$cookie2 = $query->cookie(-name=>'answers',
                          -value=>\%answers);
print $query->header(-cookie=>[$cookie1,$cookie2]);
```

To retrieve a cookie, request it by name by calling *cookie()* method without the **−value** parameter:

```
use CGI;
$query = new CGI;
%answers = $query->cookie(-name=>'answers');
# $query->cookie('answers') will work too!
```

The cookie and CGI namespaces are separate. If you have a parameter named 'answers' and a cookie named 'answers', the values retrieved by *param()* and *cookie()* are independent of each other. However, it's simple to turn a CGI parameter into a cookie, and vice-versa:

```
# turn a CGI parameter into a cookie
$c=$q->cookie(-name=>'answers',-value=>[$q->param('answers')]);
# vice-versa
$q->param(-name=>'answers',-value=>[$q->cookie('answers')]);
```

See the **cookie.cgi** example script for some ideas on how to use cookies effectively.

**NOTE:** There appear to be some (undocumented) restrictions on Netscape cookies. In Netscape 2.01, at least, I haven't been able to set more than three cookies at a time. There may also be limits on the length of cookies. If you need to store a lot of information, it's probably better to create a unique session ID, store it in a cookie, and use the session ID to locate an external file/database saved on the server's side of the connection.

**WORKING WITH NETSCAPE FRAMES**

It's possible for CGI.pm scripts to write into several browser panels and windows using Netscape's frame mechanism. There are three techniques for defining new frames programmatically:

1. Create a <Frameset> document

After writing out the HTTP header, instead of creating a standard HTML document using the *start_html()* call, create a <FRAMESET> document that defines the frames on the page. Specify your *script*(s) (with appropriate parameters) as the SRC for each of the frames.

There is no specific support for creating <FRAMESET> sections in CGI.pm, but the HTML is very simple to write. See the frame documentation in Netscape's home pages for details

```
http://home.netscape.com/assist/net_sites/frames.html
```

2. Specify the destination for the document in the HTTP header

You may provide a **–target** parameter to the *header()* method:

print $q->*header*(–target=>'ResultsWindow');

This will tell Netscape to load the output of your script into the frame named "ResultsWindow". If a frame of that name doesn't already exist, Netscape will pop up a new window and load your script's document into that. There are a number of magic names that you can use for targets. See the frame documents on Netscape's home pages for details.

3. Specify the destination for the document in the <FORM> tag

You can specify the frame to load in the FORM tag itself. With CGI.pm it looks like this:

```
print $q->startform(-target=>'ResultsWindow');
```

When your script is reinvoked by the form, its output will be loaded into the frame named "ResultsWindow". If one doesn't already exist a new window will be created.

The script "frameset.cgi" in the examples directory shows one way to create pages in which the fill-out form and the response live in side-by-side frames.

**LIMITED SUPPORT FOR CASCADING STYLE SHEETS**

CGI.pm has limited support for HTML3's cascading style sheets (css). To incorporate a stylesheet into your document, pass the *start_html()* method a **–style** parameter. The value of this parameter may be a scalar, in which case it is incorporated directly into a <STYLE> section, or it may be a hash reference. In the latter case you should provide the hash with one or more of **–src** or **–code**. **–src** points to a URL where an externally-defined stylesheet can be found. **–code** points to a scalar value to be incorporated into a <STYLE> section. Style definitions in **–code** override similarly-named ones in **–src**, hence the name "cascading."

You may also specify the type of the stylesheet by adding the optional **–type** parameter to the hash pointed to by **–style**. If not specified, the style defaults to 'text/css'.

To refer to a style within the body of your document, add the **–class** parameter to any HTML element:

```
print h1({-class=>'Fancy'},'Welcome to the Party');
```

Or define styles on the fly with the **–style** parameter:

```
print h1({-style=>'Color: red;'},'Welcome to Hell');
```

You may also use the new **span()** element to apply a style to a section of text:

```
print span({-style=>'Color: red;'},
           h1('Welcome to Hell'),
           "Where did that handbasket get to?"
           );
```

Note that you must import the ":html3" definitions to have the **span()** method available. Here's a quick and dirty example of using CSS's. See the CSS specification at http://www.w3.org/pub/WWW/TR/Wd-css-1.html for more information.

```
use CGI qw/:standard :html3/;
```

```
#here's a stylesheet incorporated directly into the page
$newStyle=<<END;
<!--
P.Tip {
    margin-right: 50pt;
    margin-left: 50pt;
    color: red;
}
P.Alert {
    font-size: 30pt;
    font-family: sans-serif;
  color: red;
}
-->
END
print header();
print start_html( -title=>'CGI with Style',
                  -style=>{-src=>'http://www.capricorn.com/style/st1.css',
                           -code=>$newStyle}
                 );
print h1('CGI with Style'),
      p({-class=>'Tip'},
        "Better read the cascading style sheet spec before playing with this!"),
      span({-style=>'color: magenta'},
           "Look Mom, no hands!",
           p(),
           "Whooo wee!"
           );
print end_html;
```

**DEBUGGING**

If you are running the script from the command line or in the perl debugger, you can pass the script a list of keywords or parameter=value pairs on the command line or from standard input (you don't have to worry about tricking your script into reading from environment variables). You can pass keywords like this:

```
your_script.pl keyword1 keyword2 keyword3
```

or this:

```
your_script.pl keyword1+keyword2+keyword3
```

or this:

```
your_script.pl name1=value1 name2=value2
```

or this:

```
your_script.pl name1=value1&name2=value2
```

or even as newline-delimited parameters on standard input.

When debugging, you can use quotes and backslashes to escape characters in the familiar shell manner, letting you place spaces and other funny characters in your parameter=value pairs:

```
your_script.pl "name1='I am a long value'" "name2=two\ words"
```

### DUMPING OUT ALL THE NAME/VALUE PAIRS

The *dump()* method produces a string consisting of all the query's name/value pairs formatted nicely as a nested list. This is useful for debugging purposes:

```
print $query->dump
```

Produces something that looks like:

```
<UL>
<LI>name1
    <UL>
    <LI>value1
    <LI>value2
    </UL>
<LI>name2
    <UL>
    <LI>value1
    </UL>
</UL>
```

You can pass a value of 'true' to *dump()* in order to get it to print the results out as plain text, suitable for incorporating into a <PRE> section.

As a shortcut, as of version 1.56 you can interpolate the entire CGI object into a string and it will be replaced with the a nice HTML dump shown above:

```
$query=new CGI;
print "<H2>Current Values</H2> $query\n";
```

## FETCHING ENVIRONMENT VARIABLES

Some of the more useful environment variables can be fetched through this interface. The methods are as follows:

**accept()** Return a list of MIME types that the remote browser accepts. If you give this method a single argument corresponding to a MIME type, as in `$query->`*accept*('text/html'), it will return a floating point value corresponding to the browser's preference for this type from 0.0 (don't want) to 1.0. Glob types (e.g. text/*) in the browser's accept list are handled correctly.

**raw_cookie()**
Returns the HTTP_COOKIE variable, an HTTP extension implemented by Netscape browsers version 1.1 and higher. Cookies have a special format, and this method call just returns the raw form (?cookie dough). See *cookie()* for ways of setting and retrieving cooked cookies.

Called with no parameters, *raw_cookie()* returns the packed cookie structure. You can separate it into individual cookies by splitting on the character sequence "; ". Called with the name of a cookie, retrieves the **unescaped** form of the cookie. You can use the regular *cookie()* method to get the names, or use the *raw_fetch()* method from the CGI::Cookie module.

**user_agent()**
Returns the HTTP_USER_AGENT variable. If you give this method a single argument, it will attempt to pattern match on it, allowing you to do something like `$query->`*user_agent*(netscape);

**path_info()**
Returns additional path information from the script URL. E.G. fetching /cgi-bin/your_script/additional/stuff will result in `$query->`*path_info()* returning "additional/stuff".

NOTE: The Microsoft Internet Information Server is broken with respect to additional path information. If you use the Perl DLL library, the IIS server will attempt to execute the additional path information as a Perl script. If you use the ordinary file associations mapping, the path information will be present in the environment, but incorrect. The best thing to do is to avoid using additional path information in CGI scripts destined for use with IIS.

**path_translated()**
As per *path_info()* but returns the additional path information translated into a physical path, e.g. ''/usr/local/etc/httpd/htdocs/additional/stuff''.

The Microsoft IIS is broken with respect to the translated path as well.

**remote_host()**
Returns either the remote host name or IP address. if the former is unavailable.

**script_name()** Return the script name as a partial URL, for self-refering scripts.

**referer()**
Return the URL of the page the browser was viewing prior to fetching your script. Not available for all browsers.

**auth_type ()**
Return the authorization/verification method in use for this script, if any.

**server_name ()**
Returns the name of the server, usually the machine's host name.

**virtual_host ()**
When using virtual hosts, returns the name of the host that the browser attempted to contact

**server_software ()**
Returns the server software and version number.

**remote_user ()**
Return the authorization/verification name used for user verification, if this script is protected.

**user_name ()**
Attempt to obtain the remote user's name, using a variety of different techniques. This only works with older browsers such as Mosaic. Netscape does not reliably report the user name!

**request_method()**
Returns the method used to access your script, usually one of 'POST', 'GET' or 'HEAD'.

## USING NPH SCRIPTS

NPH, or ''no-parsed-header'', scripts bypass the server completely by sending the complete HTTP header directly to the browser. This has slight performance benefits, but is of most use for taking advantage of HTTP extensions that are not directly supported by your server, such as server push and PICS headers.

Servers use a variety of conventions for designating CGI scripts as NPH. Many Unix servers look at the beginning of the script's name for the prefix ''nph-''. The Macintosh WebSTAR server and Microsoft's Internet Information Server, in contrast, try to decide whether a program is an NPH script by examining the first line of script output.

CGI.pm supports NPH scripts with a special NPH mode. When in this mode, CGI.pm will output the necessary extra header information when the *header()* and *redirect()* methods are called.

The Microsoft Internet Information Server requires NPH mode. As of version 2.30, CGI.pm will automatically detect when the script is running under IIS and put itself into this mode. You do not need to do this manually, although it won't hurt anything if you do.

There are a number of ways to put CGI.pm into NPH mode:

In the **use** statement
Simply add the ''−nph'' pragmato the list of symbols to be imported into your script:

```
use CGI qw(:standard -nph)
```

By calling the **nph()** method:

Call **nph()** with a non-zero parameter at any point after using CGI.pm in your program.

```
CGI->nph(1)
```

By using **–nph** parameters in the **header()** and **redirect()** statements:

```
print $q->header(-nph=>1);
```

**Server Push**

CGI.pm provides three simple functions for producing multipart documents of the type needed to implement server push. These functions were graciously provided by Ed Jordan <ed@fidalgo.net>. To import these into your namespace, you must import the ":push" set. You are also advised to put the script into NPH mode and to set $| to 1 to avoid buffering problems.

Here is a simple script that demonstrates server push:

```
#!/usr/local/bin/perl
use CGI qw/:push -nph/;
$| = 1;
print multipart_init(-boundary=>'----------------here we go!');
while (1) {
    print multipart_start(-type=>'text/plain'),
          "The current time is ",scalar(localtime),"\n",
          multipart_end;
    sleep 1;
}
```

This script initializes server push by calling **multipart_init()**. It then enters an infinite loop in which it begins a new multipart section by calling **multipart_start()**, prints the current local time, and ends a multipart section with **multipart_end()**. It then sleeps a second, and begins again.

multipart_init() multipart_init(–boundary=>$boundary);

Initialize the multipart system. The –boundary argument specifies what MIME boundary string to use to separate parts of the document. If not provided, CGI.pm chooses a reasonable boundary for you.

multipart_start()

```
multipart_start(-type=>$type)
```

Start a new part of the multipart document using the specified MIME type. If not specified, text/html is assumed.

multipart_end()

```
multipart_end()
```

End a part. You must remember to call *multipart_end()* once for each *multipart_start()*.

Users interested in server push applications should also have a look at the CGI::Push module.

**Avoiding Denial of Service Attacks**

A potential problem with CGI.pm is that, by default, it attempts to process form POSTings no matter how large they are. A wily hacker could attack your site by sending a CGI script a huge POST of many megabytes. CGI.pm will attempt to read the entire POST into a variable, growing hugely in size until it runs out of memory. While the script attempts to allocate the memory the system may slow down dramatically. This is a form of denial of service attack.

Another possible attack is for the remote user to force CGI.pm to accept a huge file upload. CGI.pm will accept the upload and store it in a temporary directory even if your script doesn't expect to receive an uploaded file. CGI.pm will delete the file automatically when it terminates, but in the meantime the remote user may have filled up the server's disk space, causing problems for other programs.

The best way to avoid denial of service attacks is to limit the amount of memory, CPU time and disk space that CGI scripts can use. Some Web servers come with built-in facilities to accomplish this. In other cases, you can use the shell *limit* or *ulimit* commands to put ceilings on CGI resource usage.

CGI.pm also has some simple built-in protections against denial of service attacks, but you must activate them before you can use them. These take the form of two global variables in the CGI name space:

**$CGI::POST_MAX**

If set to a non-negative integer, this variable puts a ceiling on the size of POSTings, in bytes. If CGI.pm detects a POST that is greater than the ceiling, it will immediately exit with an error message. This value will affect both ordinary POSTs and multipart POSTs, meaning that it limits the maximum size of file uploads as well. You should set this to a reasonably high value, such as 1 megabyte.

**$CGI::DISABLE_UPLOADS**

If set to a non-zero value, this will disable file uploads completely. Other fill-out form values will work as usual.

You can use these variables in either of two ways.

**1. On a script-by-script basis**

Set the variable at the top of the script, right after the "use" statement:

```
use CGI qw/:standard/;
use CGI::Carp 'fatalsToBrowser';
$CGI::POST_MAX=1024 * 100;  # max 100K posts
$CGI::DISABLE_UPLOADS = 1;  # no uploads
```

**2. Globally for all scripts**

Open up CGI.pm, find the definitions for `$POST_MAX` and `$DISABLE_UPLOADS`, and set them to the desired values. You'll find them towards the top of the file in a subroutine named *initialize_globals()*.

Since an attempt to send a POST larger than `$POST_MAX` bytes will cause a fatal error, you might want to use CGI::Carp to echo the fatal error message to the browser window as shown in the example above. Otherwise the remote user will see only a generic "Internal Server" error message. See the the *CGI::Carp* manpage manual page for more details.

**COMPATIBILITY WITH CGI–LIB.PL**

To make it easier to port existing programs that use cgi-lib.pl the compatibility routine "ReadParse" is provided. Porting is simple:

OLD VERSION
  require "cgi-lib.pl";
  &ReadParse;
  print "The value of the antique is `$in`{antique}.\n";

NEW VERSION
  use CGI;
  CGI::ReadParse
  print "The value of the antique is `$in`{antique}.\n";

CGI.pm's *ReadParse()* routine creates a tied variable named `%in`, which can be accessed to obtain the query variables. Like ReadParse, you can also provide your own variable. Infrequently used features of

ReadParse, such as the creation of `@in` and `$in` variables, are not supported.

Once you use ReadParse, you can retrieve the query object itself this way:

```
$q = $in{CGI};
print $q->textfield(-name=>'wow',
                    -value=>'does this really work?');
```

This allows you to start using the more interesting features of CGI.pm without rewriting your old scripts from scratch.

**AUTHOR INFORMATION**



Address bug reports and comments to: lstein@genome.wi.mit.edu

**CREDITS**

Thanks very much to:

Matt Heffron (heffron@falstaff.css.beckman.com)

James Taylor (james.taylor@srs.gov)

Scott Anguish <sanguish@digifix.com>

Mike Jewell (mlj3u@virginia.edu)

Timothy Shimmin (tes@kbs.citri.edu.au)

Joergen Haegg (jh@axis.se)

Laurent Delfosse (delfosse@csgrad1.cs.wvu.edu)

Richard Resnick (applepi1@aol.com)

Craig Bishop (csb@barwonwater.vic.gov.au)

Tony Curtis (tc@vcpc.univie.ac.at)

Tim Bunce (Tim.Bunce@ig.co.uk)

Tom Christiansen (tchrist@convex.com)

Andreas Koenig (k@franz.ww.TU−Berlin.DE)

Tim MacKenzie (Tim.MacKenzie@fulcrum.com.au)

Kevin B. Hendricks (kbhend@dogwood.tyler.wm.edu)

Stephen Dahmen (joyfire@inxpress.net)

Ed Jordan (ed@fidalgo.net)

David Alan Pisoni (david@cnation.com)

Doug MacEachern (dougm@opengroup.org)

Robin Houston (robin@oneworld.org)

...and many many more...
for suggestions and bug fixes.

**A COMPLETE EXAMPLE OF A SIMPLE FORM−BASED SCRIPT**

```
#!/usr/local/bin/perl

use CGI;

$query = new CGI;
```

```
print $query->header;
print $query->start_html("Example CGI.pm Form");
print "<H1> Example CGI.pm Form</H1>\n";
&print_prompt($query);
&do_work($query);
&print_tail;
print $query->end_html;

sub print_prompt {
   my($query) = @_;

   print $query->startform;
   print "<EM>What's your name?</EM><BR>";
   print $query->textfield('name');
   print $query->checkbox('Not my real name');

   print "<P><EM>Where can you find English Sparrows?</EM><BR>";
   print $query->checkbox_group(
                         -name=>'Sparrow locations',
                         -values=>[England,France,Spain,Asia,Hoboken],
                         -linebreak=>'yes',
                         -defaults=>[England,Asia]);

   print "<P><EM>How far can they fly?</EM><BR>",
        $query->radio_group(
                -name=>'how far',
                -values=>['10 ft','1 mile','10 miles','real far'],
                -default=>'1 mile');

   print "<P><EM>What's your favorite color?</EM>  ";
   print $query->popup_menu(-name=>'Color',
                            -values=>['black','brown','red','yellow'],
                            -default=>'red');

   print $query->hidden('Reference','Monty Python and the Holy Grail');

   print "<P><EM>What have you got there?</EM><BR>";
   print $query->scrolling_list(
                 -name=>'possessions',
                 -values=>['A Coconut','A Grail','An Icon',
                           'A Sword','A Ticket'],
                 -size=>5,
                 -multiple=>'true');

   print "<P><EM>Any parting comments?</EM><BR>";
   print $query->textarea(-name=>'Comments',
                          -rows=>10,
                          -columns=>50);

   print "<P>",$query->reset;
   print $query->submit('Action','Shout');
   print $query->submit('Action','Scream');
   print $query->endform;
   print "<HR>\n";
```

```
}

sub do_work {
   my($query) = @_;
   my(@values,$key);

   print "<H2>Here are the current settings in this form</H2>";

   foreach $key ($query->param) {
      print "<STRONG>$key</STRONG> -> ";
      @values = $query->param($key);
      print join(", ",@values),"<BR>\n";
  }
}

sub print_tail {
   print <<END;
<HR>
<ADDRESS>Lincoln D. Stein</ADDRESS><BR>
<A HREF="/">Home Page</A>
END
}
```

**BUGS**

This module has grown large and monolithic. Furthermore it's doing many things, such as handling URLs, parsing CGI input, writing HTML, etc., that are also done in the LWP modules. It should be discarded in favor of the CGI::* modules, but somehow I continue to work on it.

Note that the code is truly contorted in order to avoid spurious warnings when programs are run with the **−w** switch.

**SEE ALSO**

the *CGI::Carp* manpage, the *URI::URL* manpage, the *CGI::Request* manpage, the *CGI::MiniSvr* manpage, the *CGI::Base* manpage, the *CGI::Form* manpage, the *CGI::Apache* manpage, the *CGI::Switch* manpage, the *CGI::Push* manpage, the *CGI::Fast* manpage

**NAME**

CGI::Apache − Make things work with CGI.pm against Perl-Apache API

**SYNOPSIS**

```
require CGI::Apache;

my $q = new Apache::CGI;

$q−>print($q−>header);

#do things just like you do with CGI.pm
```

**DESCRIPTION**

When using the Perl-Apache API, your applications are faster, but the enviroment is different than CGI. This module attempts to set-up that environment as best it can.

**NOTE 1**

This module used to be named Apache::CGI. Sorry for the confusion.

**NOTE 2**

If you're going to inherit from this class, make sure to "use" it after your package declaration rather than "require" it. This is because CGI.pm does a little magic during the *import()* step in order to make autoloading work correctly.

**SEE ALSO**

*perl* (1), *Apache* (3), *CGI* (3)

**AUTHOR**

Doug MacEachern <dougm@osf.org>, hacked over by Andreas König <a.koenig@mind.de>, modified by Lincoln Stein <lt>lstein@genome.wi.mit.edu<gt>

## NAME

**CGI::Carp –** CGI routines for writing to the HTTPD (or other) error log

## SYNOPSIS

```
use CGI::Carp;

croak "We're outta here!";
confess "It was my fault: $!";
carp "It was your fault!";
warn "I'm confused";
die  "I'm dying.\n";
```

## DESCRIPTION

CGI scripts have a nasty habit of leaving warning messages in the error logs that are neither time stamped nor fully identified. Tracking down the script that caused the error is a pain. This fixes that. Replace the usual

```
use Carp;
```

with

```
use CGI::Carp
```

And the standard *warn()*, die (), *croak()*, *confess()* and *carp()* calls will automagically be replaced with functions that write out nicely time-stamped messages to the HTTP server error log.

For example:

```
[Fri Nov 17 21:40:43 1995] test.pl: I'm confused at test.pl line 3.
[Fri Nov 17 21:40:43 1995] test.pl: Got an error message: Permission denied.
[Fri Nov 17 21:40:43 1995] test.pl: I'm dying.
```

## REDIRECTING ERROR MESSAGES

By default, error messages are sent to STDERR. Most HTTPD servers direct STDERR to the server's error log. Some applications may wish to keep private error logs, distinct from the server's error log, or they may wish to direct error messages to STDOUT so that the browser will receive them.

The `carpout()` function is provided for this purpose. Since *carpout()* is not exported by default, you must import it explicitly by saying

```
use CGI::Carp qw(carpout);
```

The *carpout()* function requires one argument, which should be a reference to an open filehandle for writing errors. It should be called in a `BEGIN` block at the top of the CGI application so that compiler errors will be caught. Example:

```
BEGIN {
  use CGI::Carp qw(carpout);
  open(LOG, ">>/usr/local/cgi-logs/mycgi-log") or
    die("Unable to open mycgi-log: $!\n");
  carpout(LOG);
}
```

*carpout()* does not handle file locking on the log for you at this point.

The real STDERR is not closed -- it is moved to SAVEERR. Some servers, when dealing with CGI scripts, close their connection to the browser when the script closes STDOUT and STDERR. SAVEERR is used to prevent this from happening prematurely.

You can pass filehandles to *carpout()* in a variety of ways. The ''correct'' way according to Tom

Christiansen is to pass a reference to a filehandle GLOB:

```
carpout(\*LOG);
```

This looks weird to mere mortals however, so the following syntaxes are accepted as well:

```
carpout(LOG);
carpout(main::LOG);
carpout(main'LOG);
carpout(\LOG);
carpout(\'main::LOG');

... and so on
```

FileHandle and other objects work as well.

Use of *carpout()* is not great for performance, so it is recommended for debugging purposes or for moderate-use applications. A future version of this module may delay redirecting STDERR until one of the CGI::Carp methods is called to prevent the performance hit.

**MAKING PERL ERRORS APPEAR IN THE BROWSER WINDOW**

If you want to send fatal (die, confess) errors to the browser, ask to import the special "fatalsToBrowser" subroutine:

```
use CGI::Carp qw(fatalsToBrowser);
die "Bad error here";
```

Fatal errors will now be echoed to the browser as well as to the log. CGI::Carp arranges to send a minimal HTTP header to the browser so that even errors that occur in the early compile phase will be seen. Nonfatal errors will still be directed to the log file only (unless redirected with carpout).

**Changing the default message**

By default, the software error message is followed by a note to contact the Webmaster by e-mail with the time and date of the error. If this message is not to your liking, you can change it using the *set_message()* routine. This is not imported by default; you should import it on the *use()* line:

```
use CGI::Carp qw(fatalsToBrowser set_message);
set_message("It's not a bug, it's a feature!");
```

You may also pass in a code reference in order to create a custom error message. At run time, your code will be called with the text of the error message that caused the script to die. Example:

```
use CGI::Carp qw(fatalsToBrowser set_message);
BEGIN {
   sub handle_errors {
      my $msg = shift;
      print "<h1>Oh gosh</h1>";
      print "Got an error: $msg";
  }
  set_message(\&handle_errors);
}
```

In order to correctly intercept compile-time errors, you should call *set_message()* from within a BEGIN{} block.

**CHANGE LOG**

1.05 *carpout()* added and minor corrections by Marc Hedlund
<hedlund@best.com> on 11/26/95.

1.06 *fatalsToBrowser()* no longer aborts for fatal errors within *eval()* statements.

1.08 *set_message()* added and *carpout()* expanded to allow for FileHandle
objects.

1.09 *set_message()* now allows users to pass a code REFERENCE for
really custom error messages. croak and carp are now
exported by default. Thanks to Gunther Birznieks for the
patches.

1.10 Patch from Chris Dean (ctdean@cogit.com) to allow
module to run correctly under mod_perl.

**AUTHORS**

Lincoln D. Stein <lstein@genome.wi.mit.edu>. Feel free to redistribute this under the Perl Artistic License.

**SEE ALSO**

Carp, CGI::Base, CGI::BasePlus, CGI::Request, CGI::MiniSvr, CGI::Form, CGI::Response

**NAME**

CGI::Cookie − Interface to Netscape Cookies

**SYNOPSIS**

```
use CGI qw/:standard/;
use CGI::Cookie;

# Create new cookies and send them
$cookie1 = new CGI::Cookie(-name=>'ID',-value=>123456);
$cookie2 = new CGI::Cookie(-name=>'preferences',
                           -value=>{ font => Helvetica,
                                     size => 12 }
                           );
print header(-cookie=>[$cookie1,$cookie2]);

# fetch existing cookies
%cookies = fetch CGI::Cookie;
$id = $cookies{'ID'}->value;

# create cookies returned from an external source
%cookies = parse CGI::Cookie($ENV{COOKIE});
```

**DESCRIPTION**

CGI::Cookie is an interface to Netscape (HTTP/1.1) cookies, an innovation that allows Web servers to store persistent information on the browser's side of the connection. Although CGI::Cookie is intended to be used in conjunction with CGI.pm (and is in fact used by it internally), you can use this module independently.

For full information on cookies see

```
http://www.ics.uci.edu/pub/ietf/http/rfc2109.txt
```

**USING CGI::Cookie**

CGI::Cookie is object oriented. Each cookie object has a name and a value. The name is any scalar value. The value is any scalar or array value (associative arrays are also allowed). Cookies also have several optional attributes, including:

**1. expiration date**

The expiration date tells the browser how long to hang on to the cookie. If the cookie specifies an expiration date in the future, the browser will store the cookie information in a disk file and return it to the server every time the user reconnects (until the expiration date is reached). If the cookie species an expiration date in the past, the browser will remove the cookie from the disk file. If the expiration date is not specified, the cookie will persist only until the user quits the browser.

**2. domain**

This is a partial or complete domain name for which the cookie is valid. The browser will return the cookie to any host that matches the partial domain name. For example, if you specify a domain name of ".capricorn.com", then Netscape will return the cookie to Web servers running on any of the machines "www.capricorn.com", "ftp.capricorn.com", "feckless.capricorn.com", etc. Domain names must contain at least two periods to prevent attempts to match on top level domains like ".edu". If no domain is specified, then the browser will only return the cookie to servers on the host the cookie originated from.

**3. path**

If you provide a cookie path attribute, the browser will check it against your script's URL before returning the cookie. For example, if you specify the path "/cgi-bin", then the cookie will be returned to each of the scripts "/cgi-bin/tally.pl", "/cgi-bin/order.pl", and "/cgi-bin/customer_service/complain.pl'', but not to the script ``/cgi-private/site_admin.pl''. By default, path is set to ``/'', which causes the cookie to be sent to any CGI script on your site.

**4. secure flag**

If the ``secure'' attribute is set, the cookie will only be sent to your script if the CGI request is occurring on a secure channel, such as SSL.

### Creating New Cookies

```
$c = new CGI::Cookie(-name    =>  'foo',
                     -value   =>  'bar',
                     -expires =>  '+3M',
                     -domain  =>  '.capricorn.com',
                     -path    =>  '/cgi-bin/database'
                     -secure  =>  1
                    );
```

Create cookies from scratch with the **new** method. The **–name** and **–value** parameters are required. The name must be a scalar value. The value can be a scalar, an array reference, or a hash reference. (At some point in the future cookies will support one of the Perl object serialization protocols for full generality).

**–expires** accepts any of the relative or absolute date formats recognized by CGI.pm, for example ``+3M'' for three months in the future. See CGI.pm's documentation for details.

**–domain** points to a domain name or to a fully qualified host name. If not specified, the cookie will be returned only to the Web server that created it.

**–path** points to a partial URL on the current server. The cookie will be returned to all URLs beginning with the specified path. If not specified, it defaults to '/', which returns the cookie to all pages at your site.

**–secure** if set to a true value instructs the browser to return the cookie only when a cryptographic protocol is in use.

### Sending the Cookie to the Browser

Within a CGI script you can send a cookie to the browser by creating one or more Set-Cookie: fields in the HTTP header. Here is a typical sequence:

```
my $c = new CGI::Cookie(-name    =>  'foo',
                        -value   =>  ['bar','baz'],
                        -expires =>  '+3M');

print "Set-Cookie: $c\n";
print "Content-Type: text/html\n\n";
```

To send more than one cookie, create several Set-Cookie: fields. Alternatively, you may concatenate the cookies together with ``; `` and send them in one field.

If you are using CGI.pm, you send cookies by providing a –cookie argument to the *header()* method:

```
print header(-cookie=>$c);
```

Mod_perl users can set cookies using the request object's *header_out()* method:

```
$r->header_out('Set-Cookie',$c);
```

Internally, Cookie overloads the "" operator to call its *as_string()* method when incorporated into the HTTP header. *as_string()* turns the Cookie's internal representation into an RFC–compliant text representation. You may call *as_string()* yourself if you prefer:

```
print "Set-Cookie: ",$c->as_string,"\n";
```

### Recovering Previous Cookies

```
%cookies = fetch CGI::Cookie;
```

**fetch** returns an associative array consisting of all cookies returned by the browser. The keys of the array are the cookie names. You can iterate through the cookies this way:

```
%cookies = fetch CGI::Cookie;
foreach (keys %cookies) {
   do_something($cookies{$_});
}
```

In a scalar context, *fetch()* returns a hash reference, which may be more efficient if you are manipulating multiple cookies.

CGI.pm uses the URL escaping methods to save and restore reserved characters in its cookies. If you are trying to retrieve a cookie set by a foreign server, this escaping method may trip you up. Use *raw_fetch()* instead, which has the same semantics as *fetch()*, but performs no unescaping.

You may also retrieve cookies that were stored in some external form using the *parse()* class method:

```
$COOKIES = `cat /usr/tmp/Cookie_stash`;
%cookies = parse CGI::Cookie($COOKIES);
```

### Manipulating Cookies

Cookie objects have a series of accessor methods to get and set cookie attributes. Each accessor has a similar syntax. Called without arguments, the accessor returns the current value of the attribute. Called with an argument, the accessor changes the attribute and returns its new value.

**name()**
Get or set the cookie's name. Example:

```
$name = $c->name;
$new_name = $c->name('fred');
```

**value()**
Get or set the cookie's value. Example:

```
$value = $c->value;
@new_value = $c->value(['a','b','c','d']);
```

**value()** is context sensitive. In an array context it will return the current value of the cookie as an array. In a scalar context it will return the **first** value of a multivalued cookie.

**domain()**
Get or set the cookie's domain.

**path()**
Get or set the cookie's path.

**expires()**
Get or set the cookie's expiration time.

## AUTHOR INFORMATION

be used and modified freely, but I do request that this copyright notice remain attached to the file. You may modify this module as you wish, but if you redistribute a modified version, please attach a note listing the modifications you have made.

Address bug reports and comments to: lstein@genome.wi.mit.edu

**BUGS**

This section intentionally left blank.

**SEE ALSO**

the *CGI::Carp* manpage, the *CGI* manpage

=cut

## NAME

CGI::Fast − CGI Interface for Fast CGI

## SYNOPSIS

```
use CGI::Fast qw(:standard);
$COUNTER = 0;
while (new CGI::Fast) {
    print header;
    print start_html("Fast CGI Rocks");
    print
        h1("Fast CGI Rocks"),
        "Invocation number ",b($COUNTER++),
        " PID ",b($$),".",
        hr;
    print end_html;
}
```

## DESCRIPTION

CGI::Fast is a subclass of the CGI object created by CGI.pm. It is specialized to work well with the Open Market FastCGI standard, which greatly speeds up CGI scripts by turning them into persistently running server processes. Scripts that perform time-consuming initialization processes, such as loading large modules or opening persistent database connections, will see large performance improvements.

## OTHER PIECES OF THE PUZZLE

In order to use CGI::Fast you'll need a FastCGI−enabled Web server. Open Market's server is FastCGI−savvy. There are also freely redistributable FastCGI modules for NCSA httpd 1.5 and Apache. FastCGI−enabling modules for Microsoft Internet Information Server and Netscape Communications Server have been announced.

In addition, you'll need a version of the Perl interpreter that has been linked with the FastCGI I/O library. Precompiled binaries are available for several platforms, including DEC Alpha, HP−UX and SPARC/Solaris, or you can rebuild Perl from source with patches provided in the FastCGI developer's kit. The FastCGI Perl interpreter can be used in place of your normal Perl without ill consequences.

You can find FastCGI modules for Apache and NCSA httpd, precompiled Perl interpreters, and the FastCGI developer's kit all at URL:

```
http://www.fastcgi.com/
```

## WRITING FASTCGI PERL SCRIPTS

FastCGI scripts are persistent: one or more copies of the script are started up when the server initializes, and stay around until the server exits or they die a natural death. After performing whatever one-time initialization it needs, the script enters a loop waiting for incoming connections, processing the request, and waiting some more.

A typical FastCGI script will look like this:

```
#!/usr/local/bin/perl    # must be a FastCGI version of perl!
use CGI::Fast;
&do_some_initialization();
while ($q = new CGI::Fast) {
    &process_request($q);
}
```

Each time there's a new request, CGI::Fast returns a CGI object to your loop. The rest of the time your script waits in the call to *new()*. When the server requests that your script be terminated, *new()* will return undef. You can of course exit earlier if you choose. A new version of the script will be respawned to take its place (this may be necessary in order to avoid Perl memory leaks in long-running scripts).

CGI.pm's default CGI object mode also works. Just modify the loop this way:

```
while (new CGI::Fast) {
    &process_request;
}
```

Calls to *header()*, *start_form()*, etc. will all operate on the current request.

**INSTALLING FASTCGI SCRIPTS**

See the FastCGI developer's kit documentation for full details. On the Apache server, the following line must be added to srm.conf:

```
AddType application/x-httpd-fcgi .fcgi
```

FastCGI scripts must end in the extension .fcgi. For each script you install, you must add something like the following to srm.conf:

```
AppClass /usr/etc/httpd/fcgi-bin/file_upload.fcgi -processes 2
```

This instructs Apache to launch two copies of file_upload.fcgi at startup time.

**USING FASTCGI SCRIPTS AS CGI SCRIPTS**

Any script that works correctly as a FastCGI script will also work correctly when installed as a vanilla CGI script. However it will not see any performance benefit.

**CAVEATS**

I haven't tested this very much.

**AUTHOR INFORMATION**

be used and modified freely, but I do request that this copyright notice remain attached to the file. You may modify this module as you wish, but if you redistribute a modified version, please attach a note listing the modifications you have made.

Address bug reports and comments to: lstein@genome.wi.mit.edu

**BUGS**

This section intentionally left blank.

**SEE ALSO**

the *CGI::Carp* manpage, the *CGI* manpage

=cut

## NAME

CGI::Push − Simple Interface to Server Push

## SYNOPSIS

```
use CGI::Push qw(:standard);

do_push(−next_page=>\&next_page,
        −last_page=>\&last_page,
        −delay=>0.5);

sub next_page {
    my($q,$counter) = @_;
    return undef if $counter >= 10;
    return start_html('Test'),
           h1('Visible'),"\n",
           "This page has been called ", strong($counter)," times",
           end_html();
  }

 sub last_page {
     my($q,$counter) = @_;
     return start_html('Done'),
            h1('Finished'),
            strong($counter),' iterations.',
            end_html;
 }
```

## DESCRIPTION

CGI::Push is a subclass of the CGI object created by CGI.pm. It is specialized for server push operations, which allow you to create animated pages whose content changes at regular intervals.

You provide CGI::Push with a pointer to a subroutine that will draw one page. Every time your subroutine is called, it generates a new page. The contents of the page will be transmitted to the browser in such a way that it will replace what was there beforehand. The technique will work with HTML pages as well as with graphics files, allowing you to create animated GIFs.

## USING CGI::Push

CGI::Push adds one new method to the standard CGI suite, *do_push()*. When you call this method, you pass it a reference to a subroutine that is responsible for drawing each new page, an interval delay, and an optional subroutine for drawing the last page. Other optional parameters include most of those recognized by the CGI *header()* method.

You may call *do_push()* in the object oriented manner or not, as you prefer:

```
use CGI::Push;
$q = new CGI::Push;
$q−>do_push(−next_page=>\&draw_a_page);

    −or−

use CGI::Push qw(:standard);
do_push(−next_page=>\&draw_a_page);
```

Parameters are as follows:

-next_page

```
do_push(-next_page=>\&my_draw_routine);
```

This required parameter points to a reference to a subroutine responsible for drawing each new page. The subroutine should expect two parameters consisting of the CGI object and a counter indicating the number of times the subroutine has been called. It should return the contents of the page as an **array** of one or more items to print. It can return a false value (or an empty array) in order to abort the redrawing loop and print out the final page (if any)

```
sub my_draw_routine {
    my($q,$counter) = @_;
    return undef if $counter > 100;
    return start_html('testing'),
           h1('testing'),
           "This page called $counter times";
}
```

You are of course free to refer to create and use global variables within your draw routine in order to achieve special effects.

-last_page
This optional parameter points to a reference to the subroutine responsible for drawing the last page of the series. It is called after the –next_page routine returns a false value. The subroutine itself should have exactly the same calling conventions as the –next_page routine.

-type
This optional parameter indicates the content type of each page. It defaults to "text/html". Normally the module assumes that each page is of a homogenous MIME type. However if you provide either of the magic values "heterogeneous" or "dynamic" (the latter provided for the convenience of those who hate long parameter names), you can specify the MIME type -- and other header fields -- on a per-page basis. See "heterogeneous pages" for more details.

-delay
This indicates the delay, in seconds, between frames. Smaller delays refresh the page faster. Fractional values are allowed.

**If not specified, –delay will default to 1 second**

-cookie, –target, –expires
These have the same meaning as the like-named parameters in *CGI::header()*.

### Heterogeneous Pages

Ordinarily all pages displayed by CGI::Push share a common MIME type. However by providing a value of "heterogeneous" or "dynamic" in the *do_push()* –type parameter, you can specify the MIME type of each page on a case-by-case basis.

If you use this option, you will be responsible for producing the HTTP header for each page. Simply modify your draw routine to look like this:

```
sub my_draw_routine {
    my($q,$counter) = @_;
    return header('text/html'),   # note we're producing the header here
           start_html('testing'),
           h1('testing'),
           "This page called $counter times";
}
```

You can add any header fields that you like, but some (cookies and status fields included) may not be interpreted by the browser. One interesting effect is to display a series of pages, then, after the last page, to redirect the browser to a new URL. Because *redirect()* does b<not> work, the easiest way is with a –refresh header field, as shown below:

```
sub my_draw_routine {
    my($q,$counter) = @_;
    return undef if $counter > 10;
    return header('text/html'),   # note we're producing the header here
           start_html('testing'),
           h1('testing'),
           "This page called $counter times";
}

sub my_last_page {
    header(-refresh=>'5; URL=http://somewhere.else/finished.html',
           -type=>'text/html'),
    start_html('Moved'),
    h1('This is the last page'),
    'Goodbye!'
     hr,
     end_html;
}
```

### Changing the Page Delay on the Fly

If you would like to control the delay between pages on a page-by-page basis, call *push_delay()* from within your draw routine. *push_delay()* takes a single numeric argument representing the number of seconds you wish to delay after the current page is displayed and before displaying the next one. The delay may be fractional. Without parameters, *push_delay()* just returns the current delay.

## INSTALLING CGI::Push SCRIPTS

Server push scripts **must** be installed as no-parsed-header (NPH) scripts in order to work correctly. On Unix systems, this is most often accomplished by prefixing the script's name with "nph-". Recognition of NPH scripts happens automatically with WebSTAR and Microsoft IIS. Users of other servers should see their documentation for help.

## CAVEATS

This is a new module. It hasn't been extensively tested.

## AUTHOR INFORMATION

be used and modified freely, but I do request that this copyright notice remain attached to the file. You may modify this module as you wish, but if you redistribute a modified version, please attach a note listing the modifications you have made.

Address bug reports and comments to: lstein@genome.wi.mit.edu

## BUGS

This section intentionally left blank.

## SEE ALSO

the *CGI::Carp* manpage, the *CGI* manpage

## NAME

CGI::Switch – Try more than one constructors and return the first object available

## SYNOPSIS

```
 use CGISwitch;

  -or-

 use CGI::Switch This, That, CGI::XA, Foo, Bar, CGI;

 my $q = new CGI::Switch;
```

## DESCRIPTION

Per default the *new()* method tries to call *new()* in the three packages Apache::CGI, CGI::XA, and CGI. It returns the first CGI object it succeeds with.

The import method allows you to set up the default order of the modules to be tested.

## SEE ALSO

*perl* (1), *Apache* (3), *CGI* (3), *CGI::XA* (3)

## AUTHOR

Andreas König <a.koenig@mind.de>

## NAME

Class::Struct – declare struct-like datatypes as Perl classes

## SYNOPSIS

```
use Class::Struct;
        # declare struct, based on array:
struct( CLASS_NAME => [ ELEMENT_NAME => ELEMENT_TYPE, ... ]);
        # declare struct, based on hash:
struct( CLASS_NAME => { ELEMENT_NAME => ELEMENT_TYPE, ... });

package CLASS_NAME;
use Class::Struct;
        # declare struct, based on array, implicit class name:
struct( ELEMENT_NAME => ELEMENT_TYPE, ... );

package Myobj;
use Class::Struct;
        # declare struct with four types of elements:
struct( s => '$', a => '@', h => '%', c => 'My_Other_Class' );

$obj = new Myobj;               # constructor

                                # scalar type accessor:
$element_value = $obj->s;           # element value
$obj->s('new value');               # assign to element

                                # array type accessor:
$ary_ref = $obj->a;                 # reference to whole array
$ary_element_value = $obj->a(2);    # array element value
$obj->a(2, 'new value');            # assign to array element

                                # hash type accessor:
$hash_ref = $obj->h;                # reference to whole hash
$hash_element_value = $obj->h('x'); # hash element value
$obj->h('x', 'new value');        # assign to hash element

                                # class type accessor:
$element_value = $obj->c;           # object reference
$obj->c->method(...);               # call method of object
$obj->c(new My_Other_Class);        # assign a new object
```

## DESCRIPTION

`Class::Struct` exports a single function, `struct`. Given a list of element names and types, and optionally a class name, `struct` creates a Perl 5 class that implements a "struct-like" data structure.

The new class is given a constructor method, `new`, for creating struct objects.

Each element in the struct data has an accessor method, which is used to assign to the element and to fetch its value. The default accessor can be overridden by declaring a `sub` of the same name in the package. (See Example 2.)

Each element's type can be scalar, array, hash, or class.

### The `struct()` function

The `struct` function has three forms of parameter-list.

```
struct( CLASS_NAME => [ ELEMENT_LIST ]);
struct( CLASS_NAME => { ELEMENT_LIST });
struct( ELEMENT_LIST );
```

The first and second forms explicitly identify the name of the class being created. The third form assumes the current package name as the class name.

An object of a class created by the first and third forms is based on an array, whereas an object of a class created by the second form is based on a hash. The array-based forms will be somewhat faster and smaller; the hash-based forms are more flexible.

The class created by `struct` must not be a subclass of another class other than `UNIVERSAL`.

A function named `new` must not be explicitly defined in a class created by `struct`.

The *ELEMENT_LIST* has the form

```
NAME => TYPE, ...
```

Each name-type pair declares one element of the struct. Each element name will be defined as an accessor method unless a method by that name is explicitly defined; in the latter case, a warning is issued if the warning flag (**−w**) is set.

### Element Types and Accessor Methods

The four element types -- scalar, array, hash, and class -- are represented by strings -- `'$'`, `'@'`, `'%'`, and a class name -- optionally preceded by a `'*'`.

The accessor method provided by `struct` for an element depends on the declared type of the element.

Scalar (`'$'` or `'*$'`)
: The element is a scalar, and is initialized to `undef`.

  The accessor's argument, if any, is assigned to the element.

  If the element type is `'$'`, the value of the element (after assignment) is returned. If the element type is `'*$'`, a reference to the element is returned.

Array (`'@'` or `'*@'`)
: The element is an array, initialized to `()`.

  With no argument, the accessor returns a reference to the element's whole array.

  With one or two arguments, the first argument is an index specifying one element of the array; the second argument, if present, is assigned to the array element. If the element type is `'@'`, the accessor returns the array element value. If the element type is `'*@'`, a reference to the array element is returned.

Hash (`'%'` or `'*%'`)
: The element is a hash, initialized to `()`.

  With no argument, the accessor returns a reference to the element's whole hash.

  With one or two arguments, the first argument is a key specifying one element of the hash; the second argument, if present, is assigned to the hash element. If the element type is `'%'`, the accessor returns the hash element value. If the element type is `'*%'`, a reference to the hash element is returned.

Class (`'Class_Name'` or `'*Class_Name'`)
: The element's value must be a reference blessed to the named class or to one of its subclasses. The element is initialized to the result of calling the `new` constructor of the named class.

  The accessor's argument, if any, is assigned to the element. The accessor will `croak` if this is not an appropriate object reference.

If the element type does not start with a `'*'`, the accessor returns the element value (after assignment). If the element type starts with a `'*'`, a reference to the element itself is returned.

**EXAMPLES**

Example 1

Giving a struct element a class type that is also a struct is how structs are nested. Here, `timeval` represents a time (seconds and microseconds), and `rusage` has two elements, each of which is of type `timeval`.

```
use Class::Struct;

struct( rusage => {
    ru_utime => timeval,  # seconds
    ru_stime => timeval,  # microseconds
});

struct( timeval => [
    tv_secs  => '$',
    tv_usecs => '$',
]);

    # create an object:
my $t = new rusage;
    # $t->ru_utime and $t->ru_stime are objects of type timeval.

    # set $t->ru_utime to 100.0 sec and $t->ru_stime to 5.0 sec.
$t->ru_utime->tv_secs(100);
$t->ru_utime->tv_usecs(0);
$t->ru_stime->tv_secs(5);
$t->ru_stime->tv_usecs(0);
```

Example 2

An accessor function can be redefined in order to provide additional checking of values, etc. Here, we want the `count` element always to be nonnegative, so we redefine the `count` accessor accordingly.

```
package MyObj;
use Class::Struct;

            # declare the struct
struct ( 'MyObj', { count => '$', stuff => '%' } );

            # override the default accessor method for 'count'
sub count {
    my $self = shift;
    if ( @_ ) {
        die 'count must be nonnegative' if $_[0] < 0;
        $self->{'count'} = shift;
        warn "Too many args to count" if @_;
    }
    return $self->{'count'};
}
```

```
package main;
$x = new MyObj;
print "\$x->count(5) = ", $x->count(5), "\n";
                        # prints '$x->count(5) = 5'

print "\$x->count = ", $x->count, "\n";
                        # prints '$x->count = 5'

print "\$x->count(-5) = ", $x->count(-5), "\n";
                        # dies due to negative argument!
```

**Author and Modification History**

Renamed to `Class::Struct` and modified by Jim Miner, 1997-04-02.

```
members() function removed.
Documentation corrected and extended.
Use of struct() in a subclass prohibited.
User definition of accessor allowed.
Treatment of '*' in element types corrected.
Treatment of classes as element types corrected.
Class name to struct() made optional.
Diagnostic checks added.
```

Originally `Class::Template` by Dean Roehrich.

```
# Template.pm   --- struct/member template builder
#   12mar95
#   Dean Roehrich
#
# changes/bugs fixed since 28nov94 version:
#  - podified
# changes/bugs fixed since 21nov94 version:
#  - Fixed examples.
# changes/bugs fixed since 02sep94 version:
#  - Moved to Class::Template.
# changes/bugs fixed since 20feb94 version:
#  - Updated to be a more proper module.
#  - Added "use strict".
#  - Bug in build_methods, was using @var when @$var needed.
#  - Now using my() rather than local().
#
# Uses perl5 classes to create nested data types.
# This is offered as one implementation of Tom Christiansen's "structs.pl"
# idea.
```

**NAME**

Config − access Perl configuration information

**SYNOPSIS**

```
use Config;
if ($Config{'cc'} =~ /gcc/) {
    print "built by gcc\n";
}

use Config qw(myconfig config_sh config_vars);

print myconfig();

print config_sh();

config_vars(qw(osname archname));
```

**DESCRIPTION**

The Config module contains all the information that was available to the `Configure` program at Perl build time (over 900 values).

Shell variables from the *config.sh* file (written by Configure) are stored in the readonly-variable `%Config`, indexed by their names.

Values stored in config.sh as 'undef' are returned as undefined values. The perl `exists` function can be used to check if a named variable exists.

myconfig()
: Returns a textual summary of the major perl configuration values. See also `-V` in the `Switches` entry in the *perlrun* manpage.

config_sh()
: Returns the entire perl configuration information in the form of the original config.sh shell variable assignment script.

config_vars(@names)
: Prints to STDOUT the values of the named configuration variable. Each is printed on a separate line in the form:

```
name='value';
```

Names which are unknown are output as `name='UNKNOWN';`. See also `-V:name` in the `Switches` entry in the *perlrun* manpage.

**EXAMPLE**

Here's a more sophisticated example of using `%Config`:

```
use Config;
use strict;
```

```
my %sig_num;
my @sig_name;
unless($Config{sig_name} && $Config{sig_num}) {
    die "No sigs?";
} else {
    my @names = split ' ', $Config{sig_name};
    @sig_num{@names} = split ' ', $Config{sig_num};
    foreach (@names) {
        $sig_name[$sig_num{$_}] ||= $_;
    }
}

print "signal #17 = $sig_name[17]\n";
if ($sig_num{ALRM}) {
    print "SIGALRM is $sig_num{ALRM}\n";
}
```

**WARNING**

Because this information is not stored within the perl executable itself it is possible (but unlikely) that the information does not relate to the actual perl binary which is being used to access it.

The Config module is installed into the architecture and version specific library directory ($Config{installarchlib}) and it checks the perl version number when loaded.

The values stored in config.sh may be either single-quoted or double-quoted. Double-quoted strings are handy for those cases where you need to include escape sequences in the strings. To avoid runtime variable interpolation, any `$` and `@` characters are replaced by `\$` and `\@`, respectively. This isn't foolproof, of course, so don't embed `\$` or `\@` in double-quoted strings unless you're willing to deal with the consequences. (The slashes will end up escaped and the `$` or `@` will trigger variable interpolation)

**GLOSSARY**

Most `Config` variables are determined by the `Configure` script on platforms supported by it (which is most UNIX platforms). Some platforms have custom-made `Config` variables, and may thus not have some of the variables described below, or may have extraneous variables specific to that particular port. See the port specific documentation in such cases.

**M**

`Mcc` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the Mcc program. After Configure runs, the value is reset to a plain `Mcc` and is not useful.

**_**

`_a` From *Unix.U*:

This variable defines the extension used for ordinary libraries. For unix, it is *.a*. The . is included. Other possible values include *.lib*.

`_exe`
From *Unix.U*:

This variable defines the extension used for executable files. For unix it is empty. Other possible values include *.exe*.

`_o` From *Unix.U*:

This variable defines the extension used for object files. For unix, it is *.o*. The . is included. Other possible values include *.obj*.

**a**

`afs` From *afs.U*:

This variable is set to `true` if `AFS` (Andrew File System) is used on the system, `false` otherwise. It is possible to override this with a hint value or command line option, but you'd better know what you are doing.

`alignbytes`
From *alignbytes.U*:

This variable holds the number of bytes required to align a double. Usual values are 2, 4 and 8.

`ansi2knr`
From *ansi2knr.U*:

This variable is set if the user needs to run ansi2knr. Currently, this is not supported, so we just abort.

`aphostname`
From *d_gethname.U*:

Thie variable contains the command which can be used to compute the host name. The command is fully qualified by its absolute path, to make it safe when used by a process with super-user privileges.

`apiversion`
From *patchlevel.U*:

This is a number which identifies the lowest version of perl to have an `API` (for `XS` extensions) compatible with the present version. For example, for 5.005_01, the apiversion should be 5.005, since 5.005_01 should be binary compatible with 5.005. This should probably be incremented manually somehow, perhaps from *patchlevel.h*. For now, we'll guess maintenance subversions will retain binary compatibility.

`ar` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the ar program. After Configure runs, the value is reset to a plain `ar` and is not useful.

`archlib`
From *archlib.U*:

This variable holds the name of the directory in which the user wants to put architecture-dependent public library files for `$package`. It is most often a local directory such as */usr/local/lib*. Programs using this variable must be prepared to deal with filename expansion.

`archlibexp`
From *archlib.U*:

This variable is the same as the archlib variable, but is filename expanded at configuration time, for convenient use.

`archname`
From *archname.U*:

This variable is a short name to characterize the current architecture. It is used mainly to construct the default archlib.

`archobjs`
From *Unix.U*:

This variable defines any additional objects that must be linked in with the program on this architecture. On unix, it is usually empty. It is typically used to include emulations of unix calls or other facilities. For perl on *OS/2*, for example, this would include *os2/os2.obj*.

`awk` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the awk program. After Configure runs, the value is reset to a plain `awk` and is not useful.

**b**

`baserev`
From *baserev.U*:

The base revision level of this package, from the *.package* file.

`bash`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`bin` From *bin.U*:

This variable holds the name of the directory in which the user wants to put publicly executable images for the package in question. It is most often a local directory such as */usr/local/bin*. Programs using this variable must be prepared to deal with *~name* substitution.

`binexp`
From *bin.U*:

This is the same as the bin variable, but is filename expanded at configuration time, for use in your makefiles.

`bison`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`byacc`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the byacc program. After Configure runs, the value is reset to a plain `byacc` and is not useful.

`byteorder`
From *byteorder.U*:

This variable holds the byte order. In the following, larger digits indicate more significance. The variable byteorder is either 4321 on a big-endian machine, or 1234 on a little-endian, or 87654321 on a Cray ... or 3412 with weird order !

**c**

`c` From *n.U*:

This variable contains the \c string if that is what causes the echo command to suppress newline. Otherwise it is null. Correct usage is

```
$echo $n "prompt for a question: $c".
```

`castflags`
From *d_castneg.U*:

This variable contains a flag that precise difficulties the compiler has casting odd floating values to unsigned long:

```
0 = ok
```

```
1 = couldn't cast < 0

2 = couldn't cast >= 0x80000000

4 = couldn't cast in argument expression list
```

`cat` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the cat program. After Configure runs, the value is reset to a plain `cat` and is not useful.

`cc` From *cc.U*:

This variable holds the name of a command to execute a C compiler which can resolve multiple global references that happen to have the same name. Usual values are `cc`, `Mcc`, `cc -M`, and `gcc`.

`cccdlflags`
From *dlsrc.U*:

This variable contains any special flags that might need to be passed with `cc -c` to compile modules to be used to create a shared library that will be used for dynamic loading. For hpux, this should be +z. It is up to the makefile to use it.

`ccdlflags`
From *dlsrc.U*:

This variable contains any special flags that might need to be passed to cc to link with a shared library for dynamic loading. It is up to the makefile to use it. For sunos 4.1, it should be empty.

`ccflags`
From *ccflags.U*:

This variable contains any additional C compiler flags desired by the user. It is up to the Makefile to use this.

`cf_by`
From *cf_who.U*:

Login name of the person who ran the Configure script and answered the questions. This is used to tag both *config.sh* and *config_h.SH*.

`cf_email`
From *cf_email.U*:

Electronic mail address of the person who ran Configure. This can be used by units that require the user's e-mail, like *MailList.U*.

`cf_time`
From *cf_who.U*:

Holds the output of the `date` command when the configuration file was produced. This is used to tag both *config.sh* and *config_h.SH*.

`chgrp`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`chmod`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`chown`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`clocktype`
From *d_times.U*:

This variable holds the type returned by *times()*. It can be long, or clock_t on `BSD` sites (in which case <sys/types.h> should be included).

`comm`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the comm program. After Configure runs, the value is reset to a plain `comm` and is not useful.

`compress`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`contains`
From *contains.U*:

This variable holds the command to do a grep with a proper return status. On most sane systems it is simply `grep`. On insane systems it is a grep followed by a cat followed by a test. This variable is primarily for the use of other Configure units.

`cp` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the cp program. After Configure runs, the value is reset to a plain `cp` and is not useful.

`cpio`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`cpp` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the cpp program. After Configure runs, the value is reset to a plain `cpp` and is not useful.

`cpp_stuff`
From *cpp_stuff.U*:

This variable contains an identification of the catenation mechanism used by the C preprocessor.

`cppflags`
From *ccflags.U*:

This variable holds the flags that will be passed to the C pre- processor. It is up to the Makefile to use it.

`cpplast`
From *cppstdin.U*:

This variable has the same functionality as cppminus, only it applies to cpprun and not cppstdin.

`cppminus`
From *cppstdin.U*:

This variable contains the second part of the string which will invoke the C preprocessor on the standard input and produce to standard output. This variable will have the value − if cppstdin needs a minus to specify standard input, otherwise the value is "".

`cpprun`
From *cppstdin.U*:

This variable contains the command which will invoke a C preprocessor on standard input and put the output to stdout. It is guaranteed not to be a wrapper and may be a null string if no preprocessor can be made directly available. This preprocessor might be different from the one used by the C compiler. Don't forget to append cpplast after the preprocessor options.

`cppstdin`
From *cppstdin.U*:

This variable contains the command which will invoke the C preprocessor on standard input and put the output to stdout. It is primarily used by other Configure units that ask about preprocessor symbols.

`cryptlib`
From *d_crypt.U*:

This variable holds –lcrypt or the path to a *libcrypt.a* archive if the *crypt()* function is not defined in the standard C library. It is up to the Makefile to use this.

`csh` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the csh program. After Configure runs, the value is reset to a plain `csh` and is not useful.

**d**

`d_Gconvert`
From *d_gconvert.U*:

This variable holds what Gconvert is defined as to convert floating point numbers into strings. It could be `gconvert` or a more `complex` macro emulating gconvert with *gcvt()* or sprintf.

`d_access`
From *d_access.U*:

This variable conditionally defines `HAS_ACCESS` if the *access()* system call is available to check for access permissions using real IDs.

`d_alarm`
From *d_alarm.U*:

This variable conditionally defines the `HAS_ALARM` symbol, which indicates to the C program that the *alarm()* routine is available.

`d_archlib`
From *archlib.U*:

This variable conditionally defines `ARCHLIB` to hold the pathname of architecture-dependent library files for `$package`. If `$archlib` is the same as `$privlib`, then this is set to undef.

`d_attribut`
From *d_attribut.U*:

This variable conditionally defines `HASATTRIBUTE`, which indicates the C compiler can check for function attributes, such as printf formats.

`d_bcmp`
From *d_bcmp.U*:

This variable conditionally defines the `HAS_BCMP` symbol if the *bcmp()* routine is available to compare strings.

`d_bcopy`
From *d_bcopy.U*:

This variable conditionally defines the `HAS_BCOPY` symbol if the *bcopy()* routine is available to copy strings.

`d_bsd`
From *Guess.U*:

This symbol conditionally defines the symbol `BSD` when running on a `BSD` system.

`d_bsdgetpgrp`
From *d_getpgrp.U*:

This variable conditionally defines `USE_BSD_GETPGRP` if getpgrp needs one arguments whereas `USG` one needs none.

`d_bsdsetpgrp`
From *d_setpgrp.U*:

This variable conditionally defines `USE_BSD_SETPGRP` if setpgrp needs two arguments whereas `USG` one needs none. See also d_setpgid for a `POSIX` interface.

`d_bzero`
From *d_bzero.U*:

This variable conditionally defines the `HAS_BZERO` symbol if the *bzero()* routine is available to set memory to 0.

`d_casti32`
From *d_casti32.U*:

This variable conditionally defines CASTI32, which indicates whether the C compiler can cast large floats to 32-bit ints.

`d_castneg`
From *d_castneg.U*:

This variable conditionally defines `CASTNEG`, which indicates wether the C compiler can cast negative float to unsigned.

`d_charvspr`
From *d_vprintf.U*:

This variable conditionally defines `CHARVSPRINTF` if this system has vsprintf returning type (char*). The trend seems to be to declare it as "int *vsprintf()*".

`d_chown`
From *d_chown.U*:

This variable conditionally defines the `HAS_CHOWN` symbol, which indicates to the C program that the *chown()* routine is available.

`d_chroot`
From *d_chroot.U*:

This variable conditionally defines the `HAS_CHROOT` symbol, which indicates to the C program that the *chroot()* routine is available.

`d_chsize`
From *d_chsize.U*:

This variable conditionally defines the `CHSIZE` symbol, which indicates to the C program that the *chsize()* routine is available to truncate files. You might need a −lx to get this routine.

`d_closedir`
From *d_closedir.U*:

This variable conditionally defines `HAS_CLOSEDIR` if *closedir()* is available.

`d_const`
From *d_const.U*:

This variable conditionally defines the `HASCONST` symbol, which indicates to the C program that this C compiler knows about the const type.

`d_crypt`
From *d_crypt.U*:

This variable conditionally defines the `CRYPT` symbol, which indicates to the C program that the *crypt()* routine is available to encrypt passwords and the like.

`d_csh`
From *d_csh.U*:

This variable conditionally defines the `CSH` symbol, which indicates to the C program that the C−shell exists.

`d_cuserid`
From *d_cuserid.U*:

This variable conditionally defines the `HAS_CUSERID` symbol, which indicates to the C program that the *cuserid()* routine is available to get character login names.

`d_dbl_dig`
From *d_dbl_dig.U*:

This variable conditionally defines d_dbl_dig if this system's header files provide `DBL_DIG`, which is the number of significant digits in a double precision number.

`d_difftime`
From *d_difftime.U*:

This variable conditionally defines the `HAS_DIFFTIME` symbol, which indicates to the C program that the *difftime()* routine is available.

`d_dirnamlen`
From *i_dirent.U*:

This variable conditionally defines `DIRNAMLEN`, which indicates to the C program that the length of directory entry names is provided by a d_namelen field.

`d_dlerror`
From *d_dlerror.U*:

This variable conditionally defines the `HAS_DLERROR` symbol, which indicates to the C program that the *dlerror()* routine is available.

`d_dlopen`
From *d_dlopen.U*:

This variable conditionally defines the `HAS_DLOPEN` symbol, which indicates to the C program that the *dlopen()* routine is available.

`d_dlsymun`
From *d_dlsymun.U*:

This variable conditionally defines `DLSYM_NEEDS_UNDERSCORE`, which indicates that we need to prepend an underscore to the symbol name before calling *dlsym()*.

`d_dosuid`
From *d_dosuid.U*:

This variable conditionally defines the symbol `DOSUID`, which tells the C program that it should insert setuid emulation code on hosts which have setuid #! scripts disabled.

`d_dup2`
From *d_dup2.U*:

This variable conditionally defines HAS_DUP2 if *dup2()* is available to duplicate file descriptors.

`d_endgrent`
From *d_endgrent.U*:

This variable conditionally defines the `HAS_ENDGRENT` symbol, which indicates to the C program that the *endgrent()* routine is available for sequential access of the group database.

`d_endhent`
From *d_endhent.U*:

This variable conditionally defines `HAS_ENDHOSTENT` if *endhostent()* is available to close whatever was being used for host queries.

`d_endnent`
From *d_endnent.U*:

This variable conditionally defines `HAS_ENDNETENT` if *endnetent()* is available to close whatever was being used for network queries.

`d_endpent`
From *d_endpent.U*:

This variable conditionally defines `HAS_ENDPROTOENT` if *endprotoent()* is available to close whatever was being used for protocol queries.

`d_endpwent`
From *d_endpwent.U*:

This variable conditionally defines the `HAS_ENDPWENT` symbol, which indicates to the C program that the *endpwent()* routine is available for sequential access of the passwd database.

`d_endsent`
From *d_endsent.U*:

This variable conditionally defines `HAS_ENDSERVENT` if *endservent()* is available to close whatever was being used for service queries.

`d_eofnblk`
From *nblock_io.U*:

This variable conditionally defines `EOF_NONBLOCK` if `EOF` can be seen when reading from a non-blocking *I/O* source.

`d_eunice`
From *Guess.U*:

This variable conditionally defines the symbols `EUNICE` and `VAX`, which alerts the C program that it must deal with ideosyncracies of `VMS`.

`d_fchmod`
From *d_fchmod.U*:

This variable conditionally defines the `HAS_FCHMOD` symbol, which indicates to the C program that the *fchmod()* routine is available to change mode of opened files.

`d_fchown`
From *d_fchown.U*:

This variable conditionally defines the `HAS_FCHOWN` symbol, which indicates to the C program that the *fchown()* routine is available to change ownership of opened files.

`d_fcntl`
From *d_fcntl.U*:

This variable conditionally defines the `HAS_FCNTL` symbol, and indicates whether the *fcntl()* function exists

`d_fd_macros`
From *d_fd_set.U*:

This variable contains the eventual value of the `HAS_FD_MACROS` symbol, which indicates if your C compiler knows about the macros which manipulate an fd_set.

`d_fd_set`
From *d_fd_set.U*:

This variable contains the eventual value of the `HAS_FD_SET` symbol, which indicates if your C compiler knows about the fd_set typedef.

`d_fds_bits`
From *d_fd_set.U*:

This variable contains the eventual value of the `HAS_FDS_BITS` symbol, which indicates if your fd_set typedef contains the fds_bits member. If you have an fd_set typedef, but the dweebs who installed it did a half-fast job and neglected to provide the macros to manipulate an fd_set, `HAS_FDS_BITS` will let us know how to fix the gaffe.

`d_fgetpos`
From *d_fgetpos.U*:

This variable conditionally defines `HAS_FGETPOS` if *fgetpos()* is available to get the file position indicator.

`d_flexfnam`
From *d_flexfnam.U*:

This variable conditionally defines the `FLEXFILENAMES` symbol, which indicates that the system supports filenames longer than 14 characters.

`d_flock`
From *d_flock.U*:

This variable conditionally defines `HAS_FLOCK` if *flock()* is available to do file locking.

`d_fork`
From *d_fork.U*:

This variable conditionally defines the `HAS_FORK` symbol, which indicates to the C program that the *fork()* routine is available.

`d_fpathconf`
From *d_pathconf.U*:

This variable conditionally defines the `HAS_FPATHCONF` symbol, which indicates to the C program that the *pathconf()* routine is available to determine file-system related limits and options associated with a given open file descriptor.

`d_fsetpos`
From *d_fsetpos.U*:

This variable conditionally defines `HAS_FSETPOS` if *fsetpos()* is available to set the file position indicator.

`d_ftime`
From *d_ftime.U*:

This variable conditionally defines the `HAS_FTIME` symbol, which indicates that the *ftime()* routine exists. The *ftime()* routine is basically a sub-second accuracy clock.

`d_getgrent`
From *d_getgrent.U*:

This variable conditionally defines the `HAS_GETGRENT` symbol, which indicates to the C program that the *getgrent()* routine is available for sequential access of the group database.

`d_getgrps`
From *d_getgrps.U*:

This variable conditionally defines the `HAS_GETGROUPS` symbol, which indicates to the C program that the *getgroups()* routine is available to get the list of process groups.

`d_gethbyaddr`
From *d_gethbyad.U*:

This variable conditionally defines the `HAS_GETHOSTBYADDR` symbol, which indicates to the C program that the *gethostbyaddr()* routine is available to look up hosts by their `IP` addresses.

`d_gethbyname`
From *d_gethbynm.U*:

This variable conditionally defines the `HAS_GETHOSTBYNAME` symbol, which indicates to the C program that the *gethostbyname()* routine is available to look up host names in some data base or other.

`d_gethent`
From *d_gethent.U*:

This variable conditionally defines `HAS_GETHOSTENT` if *gethostent()* is available to look up host names in some data base or another.

`d_gethname`
From *d_gethname.U*:

This variable conditionally defines the `HAS_GETHOSTNAME` symbol, which indicates to the C program that the *gethostname()* routine may be used to derive the host name.

`d_gethostprotos`
From *d_gethostprotos.U*:

This variable conditionally defines the `HAS_GETHOST_PROTOS` symbol, which indicates to the C program that <netdb.h> supplies prototypes for the various gethost*() functions. See also *netdbtype.U* for probing for various netdb types.

`d_getlogin`
From *d_getlogin.U*:

This variable conditionally defines the `HAS_GETLOGIN` symbol, which indicates to the C program that the *getlogin()* routine is available to get the login name.

`d_getnbyaddr`
From *d_getnbyad.U*:

This variable conditionally defines the `HAS_GETNETBYADDR` symbol, which indicates to the C program that the *getnetbyaddr()* routine is available to look up networks by their `IP` addresses.

`d_getnbyname`
From *d_getnbynm.U*:

This variable conditionally defines the `HAS_GETNETBYNAME` symbol, which indicates to the C program that the *getnetbyname()* routine is available to look up networks by their names.

`d_getnent`
From *d_getnent.U*:

This variable conditionally defines `HAS_GETNETENT` if *getnetent()* is available to look up network names in some data base or another.

`d_getnetprotos`
From *d_getnetprotos.U*:

This variable conditionally defines the `HAS_GETNET_PROTOS` symbol, which indicates to the C program that <netdb.h> supplies prototypes for the various getnet*() functions. See also *netdbtype.U* for probing for various netdb types.

`d_getpbyname`
From *d_getprotby.U*:

This variable conditionally defines the `HAS_GETPROTOBYNAME` symbol, which indicates to the C program that the *getprotobyname()* routine is available to look up protocols by their name.

`d_getpbynumber`
From *d_getprotby.U*:

This variable conditionally defines the `HAS_GETPROTOBYNUMBER` symbol, which indicates to the C program that the *getprotobynumber()* routine is available to look up protocols by their number.

`d_getpent`
From *d_getpent.U*:

This variable conditionally defines `HAS_GETPROTOENT` if *getprotoent()* is available to look up protocols in some data base or another.

`d_getpgid`
From *d_getpgid.U*:

This variable conditionally defines the `HAS_GETPGID` symbol, which indicates to the C program that the *getpgid* (pid) function is available to get the process group id.

`d_getpgrp2`
From *d_getpgrp2.U*:

This variable conditionally defines the HAS_GETPGRP2 symbol, which indicates to the C program that the *getpgrp2()* (as in *DG/*UX) routine is available to get the current process group.

`d_getpgrp`
From *d_getpgrp.U*:

This variable conditionally defines `HAS_GETPGRP` if *getpgrp()* is available to get the current process group.

`d_getppid`
From *d_getppid.U*:

This variable conditionally defines the `HAS_GETPPID` symbol, which indicates to the C program that the *getppid()* routine is available to get the parent process `ID`.

`d_getprior`
From *d_getprior.U*:

This variable conditionally defines `HAS_GETPRIORITY` if *getpriority()* is available to get a process's priority.

`d_getprotoprotos`
From *d_getprotoprotos.U*:

This variable conditionally defines the `HAS_GETPROTO_PROTOS` symbol, which indicates to the C program that <netdb.h> supplies prototypes for the various getproto*() functions. See also *netdbtype.U* for probing for various netdb types.

`d_getpwent`
From *d_getpwent.U*:

This variable conditionally defines the `HAS_GETPWENT` symbol, which indicates to the C program that the *getpwent()* routine is available for sequential access of the passwd database.

`d_getsbyname`
From *d_getsrvby.U*:

This variable conditionally defines the `HAS_GETSERVBYNAME` symbol, which indicates to the C program that the *getservbyname()* routine is available to look up services by their name.

`d_getsbyport`
From *d_getsrvby.U*:

This variable conditionally defines the `HAS_GETSERVBYPORT` symbol, which indicates to the C program that the *getservbyport()* routine is available to look up services by their port.

`d_getsent`
From *d_getsent.U*:

This variable conditionally defines `HAS_GETSERVENT` if *getservent()* is available to look up network services in some data base or another.

`d_getservprotos`
From *d_getservprotos.U*:

This variable conditionally defines the `HAS_GETSERV_PROTOS` symbol, which indicates to the C program that <netdb.h> supplies prototypes for the various getserv*() functions. See also *netdbtype.U* for probing for various netdb types.

`d_gettimeod`
From *d_ftime.U*:

This variable conditionally defines the `HAS_GETTIMEOFDAY` symbol, which indicates that the *gettimeofday()* system call exists (to obtain a sub-second accuracy clock). You should probably include <sys/resource.h>.

`d_gnulibc`
From *d_gnulibc.U*:

Defined if we're dealing with the `GNU` C Library.

`d_grpasswd`
From *i_grp.U*:

This variable conditionally defines `GRPASSWD`, which indicates that struct group in <grp.h> contains gr_passwd.

`d_htonl`
From *d_htonl.U*:

This variable conditionally defines `HAS_HTONL` if *htonl()* and its friends are available to do network order byte swapping.

`d_index`
From *d_strchr.U*:

This variable conditionally defines `HAS_INDEX` if *index()* and *rindex()* are available for string searching.

`d_inetaton`
From *d_inetaton.U*:

This variable conditionally defines the `HAS_INET_ATON` symbol, which indicates to the C program that the *inet_aton()* function is available to parse `IP` address `dotted-quad` strings.

`d_isascii`
From *d_isascii.U*:

This variable conditionally defines the `HAS_ISASCII` constant, which indicates to the C program that *isascii()* is available.

`d_killpg`
From *d_killpg.U*:

This variable conditionally defines the `HAS_KILLPG` symbol, which indicates to the C program that the *killpg()* routine is available to kill process groups.

`d_lchown`
From *d_lchown.U*:

This variable conditionally defines the `HAS_LCHOWN` symbol, which indicates to the C program that the *lchown()* routine is available to operate on a symbolic link (instead of following the link).

`d_link`
From *d_link.U*:

This variable conditionally defines `HAS_LINK` if *link()* is available to create hard links.

`d_locconv`
From *d_locconv.U*:

This variable conditionally defines `HAS_LOCALECONV` if *localeconv()* is available for numeric and monetary formatting conventions.

`d_lockf`
From *d_lockf.U*:

This variable conditionally defines `HAS_LOCKF` if *lockf()* is available to do file locking.

`d_longdbl`
From *d_longdbl.U*:

This variable conditionally defines `HAS_LONG_DOUBLE` if the long double type is supported.

`d_longlong`
From *d_longlong.U*:

This variable conditionally defines `HAS_LONG_LONG` if the long long type is supported.

`d_lstat`
From *d_lstat.U*:

This variable conditionally defines `HAS_LSTAT` if *lstat()* is available to do file stats on symbolic links.

`d_mblen`
From *d_mblen.U*:

This variable conditionally defines the `HAS_MBLEN` symbol, which indicates to the C program that the *mblen()* routine is available to find the number of bytes in a multibye character.

`d_mbstowcs`
From *d_mbstowcs.U*:

This variable conditionally defines the `HAS_MBSTOWCS` symbol, which indicates to the C program that the *mbstowcs()* routine is available to convert a multibyte string into a wide character string.

`d_mbtowc`
From *d_mbtowc.U*:

This variable conditionally defines the `HAS_MBTOWC` symbol, which indicates to the C program that the *mbtowc()* routine is available to convert multibyte to a wide character.

`d_memcmp`
From *d_memcmp.U*:

This variable conditionally defines the `HAS_MEMCMP` symbol, which indicates to the C program that the *memcmp()* routine is available to compare blocks of memory.

`d_memcpy`
From *d_memcpy.U*:

This variable conditionally defines the `HAS_MEMCPY` symbol, which indicates to the C program that the *memcpy()* routine is available to copy blocks of memory.

`d_memmove`
From *d_memmove.U*:

This variable conditionally defines the `HAS_MEMMOVE` symbol, which indicates to the C program that the *memmove()* routine is available to copy potentatially overlapping blocks of memory.

`d_memset`
From *d_memset.U*:

This variable conditionally defines the `HAS_MEMSET` symbol, which indicates to the C program that the *memset()* routine is available to set blocks of memory.

`d_mkdir`
From *d_mkdir.U*:

This variable conditionally defines the `HAS_MKDIR` symbol, which indicates to the C program that the *mkdir()* routine is available to create *directories.*.

`d_mkfifo`
From *d_mkfifo.U*:

This variable conditionally defines the `HAS_MKFIFO` symbol, which indicates to the C program that the *mkfifo()* routine is available.

`d_mktime`
From *d_mktime.U*:

This variable conditionally defines the `HAS_MKTIME` symbol, which indicates to the C program that the *mktime()* routine is available.

`d_msg`
From *d_msg.U*:

This variable conditionally defines the `HAS_MSG` symbol, which indicates that the entire msg*(2) library is present.

`d_msgctl`
From *d_msgctl.U*:

This variable conditionally defines the `HAS_MSGCTL` symbol, which indicates to the C program that the *msgctl()* routine is available.

`d_msgget`
From *d_msgget.U*:

This variable conditionally defines the `HAS_MSGGET` symbol, which indicates to the C program that the *msgget()* routine is available.

`d_msgrcv`
From *d_msgrcv.U*:

This variable conditionally defines the `HAS_MSGRCV` symbol, which indicates to the C program that the *msgrcv()* routine is available.

`d_msgsnd`
From *d_msgsnd.U*:

This variable conditionally defines the `HAS_MSGSND` symbol, which indicates to the C program that the *msgsnd()* routine is available.

`d_mymalloc`
From *mallocsrc.U*:

This variable conditionally defines `MYMALLOC` in case other parts of the source want to take special action if `MYMALLOC` is used. This may include different sorts of profiling or error detection.

`d_nice`
From *d_nice.U*:

This variable conditionally defines the `HAS_NICE` symbol, which indicates to the C program that the *nice()* routine is available.

`d_oldpthreads`
From *usethreads.U*:

This variable conditionally defines the `OLD_PTHREADS_API` symbol, and indicates that Perl should be built to use the old draft `POSIX` threads `API`. This is only potneially meaningful if usethreads is set.

`d_oldsock`
From *d_socket.U*:

This variable conditionally defines the `OLDSOCKET` symbol, which indicates that the `BSD` socket interface is based on 4.1c and not 4.2.

`d_open3`
From *d_open3.U*:

This variable conditionally defines the HAS_OPEN3 manifest constant, which indicates to the C program that the 3 argument version of the *open* (2) function is available.

`d_pathconf`
From *d_pathconf.U*:

This variable conditionally defines the `HAS_PATHCONF` symbol, which indicates to the C program that the *pathconf()* routine is available to determine file-system related limits and options associated with a given filename.

`d_pause`
From *d_pause.U*:

This variable conditionally defines the `HAS_PAUSE` symbol, which indicates to the C program that the *pause()* routine is available to suspend a process until a signal is received.

`d_phostname`
From *d_gethname.U*:

This variable conditionally defines the `PHOSTNAME` symbol, which contains the shell command which, when fed to *popen()*, may be used to derive the host name.

`d_pipe`
From *d_pipe.U*:

This variable conditionally defines the `HAS_PIPE` symbol, which indicates to the C program that the *pipe()* routine is available to create an inter-process channel.

`d_poll`
From *d_poll.U*:

This variable conditionally defines the `HAS_POLL` symbol, which indicates to the C program that the *poll()* routine is available to poll active file descriptors.

`d_portable`
From *d_portable.U*:

This variable conditionally defines the `PORTABLE` symbol, which indicates to the C program that it should not assume that it is running on the machine it was compiled on.

`d_pthread_yield`
From *d_pthread_y.U*:

This variable conditionally defines the `HAS_PTHREAD_YIELD` symbol if the pthread_yield routine is available to yield the execution of the current thread.

`d_pthreads_created_joinable`
From *d_pthreadj.U*:

This variable conditionally defines the `PTHREADS_CREATED_JOINABLE` symbol if pthreads are created in the joinable (aka undetached) state.

`d_pwage`
From *i_pwd.U*:

This variable conditionally defines `PWAGE`, which indicates that struct passwd contains pw_age.

`d_pwchange`
From *i_pwd.U*:

This variable conditionally defines `PWCHANGE`, which indicates that struct passwd contains pw_change.

`d_pwclass`
From *i_pwd.U*:

This variable conditionally defines `PWCLASS`, which indicates that struct passwd contains pw_class.

`d_pwcomment`
From *i_pwd.U*:

This variable conditionally defines `PWCOMMENT`, which indicates that struct passwd contains pw_comment.

`d_pwexpire`
From *i_pwd.U*:

This variable conditionally defines `PWEXPIRE`, which indicates that struct passwd contains pw_expire.

`d_pwgecos`
From *i_pwd.U*:

This variable conditionally defines `PWGECOS`, which indicates that struct passwd contains pw_gecos.

`d_pwpasswd`
From *i_pwd.U*:

This variable conditionally defines `PWPASSWD`, which indicates that struct passwd contains pw_passwd.

`d_pwquota`
From *i_pwd.U*:

This variable conditionally defines `PWQUOTA`, which indicates that struct passwd contains pw_quota.

`d_readdir`
From *d_readdir.U*:

This variable conditionally defines `HAS_READDIR` if *readdir()* is available to read directory entries.

`d_readlink`
From *d_readlink.U*:

This variable conditionally defines the `HAS_READLINK` symbol, which indicates to the C program that the *readlink()* routine is available to read the value of a symbolic link.

`d_rename`
From *d_rename.U*:

This variable conditionally defines the `HAS_RENAME` symbol, which indicates to the C program that the *rename()* routine is available to rename files.

`d_rewinddir`
From *d_readdir.U*:

This variable conditionally defines `HAS_REWINDDIR` if *rewinddir()* is available.

`d_rmdir`
From *d_rmdir.U*:

This variable conditionally defines `HAS_RMDIR` if *rmdir()* is available to remove directories.

`d_safebcpy`
From *d_safebcpy.U*:

This variable conditionally defines the `HAS_SAFE_BCOPY` symbol if the *bcopy()* routine can do overlapping copies.

`d_safemcpy`
From *d_safemcpy.U*:

This variable conditionally defines the `HAS_SAFE_MEMCPY` symbol if the *memcpy()* routine can do overlapping copies.

`d_sanemcmp`
From *d_sanemcmp.U*:

This variable conditionally defines the `HAS_SANE_MEMCMP` symbol if the *memcpy()* routine is available and can be used to compare relative magnitudes of chars with their high bits set.

`d_sched_yield`
From *d_pthread_y.U*:

This variable conditionally defines the `HAS_SCHED_YIELD` symbol if the sched_yield routine is available to yield the execution of the current thread.

`d_seekdir`
From *d_readdir.U*:

This variable conditionally defines `HAS_SEEKDIR` if *seekdir()* is available.

`d_select`
From *d_select.U*:

This variable conditionally defines `HAS_SELECT` if *select()* is available to select active file descriptors. A <sys/time.h> inclusion may be necessary for the timeout field.

`d_sem`
From *d_sem.U*:

This variable conditionally defines the `HAS_SEM` symbol, which indicates that the entire sem*(2) library is present.

`d_semctl`
From *d_semctl.U*:

This variable conditionally defines the `HAS_SEMCTL` symbol, which indicates to the C program that the *semctl()* routine is available.

`d_semctl_semid_ds`
From *d_union_senum.U*:

This variable conditionally defines `USE_SEMCTL_SEMID_DS`, which indicates that struct semid_ds * is to be used for semctl `IPC_STAT`.

`d_semctl_semun`
From *d_union_senum.U*:

This variable conditionally defines `USE_SEMCTL_SEMUN`, which indicates that union semun is to be used for semctl `IPC_STAT`.

`d_semget`
From *d_semget.U*:

This variable conditionally defines the `HAS_SEMGET` symbol, which indicates to the C program that the *semget()* routine is available.

`d_semop`
From *d_semop.U*:

This variable conditionally defines the `HAS_SEMOP` symbol, which indicates to the C program that the *semop()* routine is available.

`d_setegid`
From *d_setegid.U*:

This variable conditionally defines the `HAS_SETEGID` symbol, which indicates to the C program that the *setegid()* routine is available to change the effective gid of the current program.

`d_seteuid`
From *d_seteuid.U*:

This variable conditionally defines the `HAS_SETEUID` symbol, which indicates to the C program that the *seteuid()* routine is available to change the effective uid of the current program.

`d_setgrent`
From *d_setgrent.U*:

This variable conditionally defines the `HAS_SETGRENT` symbol, which indicates to the C program that the *setgrent()* routine is available for initializing sequential access to the group database.

`d_setgrps`
From *d_setgrps.U*:

This variable conditionally defines the `HAS_SETGROUPS` symbol, which indicates to the C program that the *setgroups()* routine is available to set the list of process groups.

`d_sethent`
From *d_sethent.U*:

This variable conditionally defines `HAS_SETHOSTENT` if *sethostent()* is available.

`d_setlinebuf`
From *d_setlnbuf.U*:

This variable conditionally defines the `HAS_SETLINEBUF` symbol, which indicates to the C program that the *setlinebuf()* routine is available to change stderr or stdout from block-buffered or unbuffered to a line-buffered mode.

`d_setlocale`
From *d_setlocale.U*:

This variable conditionally defines `HAS_SETLOCALE` if *setlocale()* is available to handle locale-specific ctype implementations.

`d_setnent`
From *d_setnent.U*:

This variable conditionally defines `HAS_SETNETENT` if *setnetent()* is available.

`d_setpent`
From *d_setpent.U*:

This variable conditionally defines `HAS_SETPROTOENT` if *setprotoent()* is available.

`d_setpgid`
From *d_setpgid.U*:

This variable conditionally defines the `HAS_SETPGID` symbol if the *setpgid* (pid, gpid) function is available to set process group `ID`.

`d_setpgrp2`
From *d_setpgrp2.U*:

This variable conditionally defines the HAS_SETPGRP2 symbol, which indicates to the C program that the *setpgrp2()* (as in *DG/*`UX`) routine is available to set the current process group.

`d_setpgrp`
From *d_setpgrp.U*:

This variable conditionally defines `HAS_SETPGRP` if *setpgrp()* is available to set the current process group.

`d_setprior`
From *d_setprior.U*:

This variable conditionally defines `HAS_SETPRIORITY` if *setpriority()* is available to set a process's priority.

`d_setpwent`
From *d_setpwent.U*:

This variable conditionally defines the `HAS_SETPWENT` symbol, which indicates to the C program that the *setpwent()* routine is available for initializing sequential access to the passwd database.

`d_setregid`
From *d_setregid.U*:

This variable conditionally defines `HAS_SETREGID` if *setregid()* is available to change the real and effective gid of the current process.

`d_setresgid`
From *d_setregid.U*:

This variable conditionally defines `HAS_SETRESGID` if *setresgid()* is available to change the real, effective and saved gid of the current process.

`d_setresuid`
From *d_setreuid.U*:

This variable conditionally defines `HAS_SETREUID` if *setresuid()* is available to change the real, effective and saved uid of the current process.

`d_setreuid`
From *d_setreuid.U*:

This variable conditionally defines `HAS_SETREUID` if *setreuid()* is available to change the real and effective uid of the current process.

`d_setrgid`
From *d_setrgid.U*:

This variable conditionally defines the `HAS_SETRGID` symbol, which indicates to the C program that the *setrgid()* routine is available to change the real gid of the current program.

`d_setruid`
From *d_setruid.U*:

This variable conditionally defines the `HAS_SETRUID` symbol, which indicates to the C program that the *setruid()* routine is available to change the real uid of the current program.

`d_setsent`
From *d_setsent.U*:

This variable conditionally defines `HAS_SETSERVENT` if *setservent()* is available.

`d_setsid`
From *d_setsid.U*:

This variable conditionally defines `HAS_SETSID` if *setsid()* is available to set the process group `ID`.

`d_setvbuf`
From *d_setvbuf.U*:

This variable conditionally defines the `HAS_SETVBUF` symbol, which indicates to the C program that the *setvbuf()* routine is available to change buffering on an open stdio stream.

`d_sfio`
From *d_sfio.U*:

This variable conditionally defines the `USE_SFIO` symbol, and indicates whether sfio is available (and should be used).

`d_shm`
From *d_shm.U*:

This variable conditionally defines the `HAS_SHM` symbol, which indicates that the entire shm*(2) library is present.

`d_shmat`
From *d_shmat.U*:

This variable conditionally defines the `HAS_SHMAT` symbol, which indicates to the C program that the *shmat()* routine is available.

`d_shmatprototype`
From *d_shmat.U*:

This variable conditionally defines the `HAS_SHMAT_PROTOTYPE` symbol, which indicates that *sys/shm.h* has a prototype for shmat.

`d_shmctl`
From *d_shmctl.U*:

This variable conditionally defines the `HAS_SHMCTL` symbol, which indicates to the C program that the *shmctl()* routine is available.

`d_shmdt`
From *d_shmdt.U*:

This variable conditionally defines the `HAS_SHMDT` symbol, which indicates to the C program that the *shmdt()* routine is available.

`d_shmget`
From *d_shmget.U*:

This variable conditionally defines the `HAS_SHMGET` symbol, which indicates to the C program that the *shmget()* routine is available.

`d_sigaction`
From *d_sigaction.U*:

This variable conditionally defines the `HAS_SIGACTION` symbol, which indicates that the Vr4 *sigaction()* routine is available.

`d_sigsetjmp`
From *d_sigsetjmp.U*:

This variable conditionally defines the `HAS_SIGSETJMP` symbol, which indicates that the *sigsetjmp()* routine is available to call *setjmp()* and optionally save the process's signal mask.

`d_socket`
From *d_socket.U*:

This variable conditionally defines `HAS_SOCKET`, which indicates that the `BSD` socket interface is supported.

`d_sockpair`
From *d_socket.U*:

This variable conditionally defines the `HAS_SOCKETPAIR` symbol, which indicates that the `BSD` *socketpair()* is supported.

`d_statblks`
From *d_statblks.U*:

This variable conditionally defines `USE_STAT_BLOCKS` if this system has a stat structure declaring st_blksize and st_blocks.

`d_stdio_cnt_lval`
From *d_stdstdio.U*:

This variable conditionally defines `STDIO_CNT_LVALUE` if the `FILE_cnt` macro can be used as an lvalue.

`d_stdio_ptr_lval`
From *d_stdstdio.U*:

This variable conditionally defines `STDIO_PTR_LVALUE` if the `FILE_ptr` macro can be used as an lvalue.

`d_stdiobase`
From *d_stdstdio.U*:

This variable conditionally defines `USE_STDIO_BASE` if this system has a `FILE` structure declaring a usable _base field (or equivalent) in *stdio.h*.

`d_stdstdio`
From *d_stdstdio.U*:

This variable conditionally defines `USE_STDIO_PTR` if this system has a `FILE` structure declaring usable _ptr and _cnt fields (or equivalent) in *stdio.h*.

`d_strchr`
From *d_strchr.U*:

This variable conditionally defines `HAS_STRCHR` if *strchr()* and *strrchr()* are available for string searching.

`d_strcoll`
From *d_strcoll.U*:

This variable conditionally defines `HAS_STRCOLL` if *strcoll()* is available to compare strings using collating information.

`d_strctcpy`
From *d_strctcpy.U*:

This variable conditionally defines the `USE_STRUCT_COPY` symbol, which indicates to the C program that this C compiler knows how to copy structures.

`d_strerrm`
From *d_strerror.U*:

This variable holds what Strerrr is defined as to translate an error code condition into an error message string. It could be `strerror` or a more `complex` macro emulating strrror with sys_errlist[], or the `unknown` string when both strerror and sys_errlist are missing.

`d_strerror`
From *d_strerror.U*:

This variable conditionally defines `HAS_STRERROR` if *strerror()* is available to translate error numbers to strings.

`d_strtod`
From *d_strtod.U*:

This variable conditionally defines the `HAS_STRTOD` symbol, which indicates to the C program that the *strtod()* routine is available to provide better numeric string conversion than *atof()*.

`d_strtol`
From *d_strtol.U*:

This variable conditionally defines the `HAS_STRTOL` symbol, which indicates to the C program that the *strtol()* routine is available to provide better numeric string conversion than *atoi()* and friends.

`d_strtoul`
From *d_strtoul.U*:

This variable conditionally defines the `HAS_STRTOUL` symbol, which indicates to the C program that the *strtoul()* routine is available to provide conversion of strings to unsigned long.

`d_strxfrm`
From *d_strxfrm.U*:

This variable conditionally defines `HAS_STRXFRM` if *strxfrm()* is available to transform strings.

`d_suidsafe`
From *d_dosuid.U*:

This variable conditionally defines `SETUID_SCRIPTS_ARE_SECURE_NOW` if setuid scripts can be secure. This test looks in */dev/fd/*.

`d_symlink`
From *d_symlink.U*:

This variable conditionally defines the `HAS_SYMLINK` symbol, which indicates to the C program that the *symlink()* routine is available to create symbolic links.

`d_syscall`
From *d_syscall.U*:

This variable conditionally defines `HAS_SYSCALL` if *syscall()* is available call arbitrary system calls.

`d_sysconf`
From *d_sysconf.U*:

This variable conditionally defines the `HAS_SYSCONF` symbol, which indicates to the C program that the *sysconf()* routine is available to determine system related limits and options.

`d_sysernlst`
From *d_strerror.U*:

This variable conditionally defines `HAS_SYS_ERRNOLIST` if sys_errnolist[] is available to translate error numbers to the symbolic name.

`d_syserrlst`
From *d_strerror.U*:

This variable conditionally defines `HAS_SYS_ERRLIST` if sys_errlist[] is available to translate error numbers to strings.

`d_system`
From *d_system.U*:

This variable conditionally defines `HAS_SYSTEM` if *system()* is available to issue a shell command.

`d_tcgetpgrp`
From *d_tcgtpgrp.U*:

This variable conditionally defines the `HAS_TCGETPGRP` symbol, which indicates to the C program that the *tcgetpgrp()* routine is available. to get foreground process group `ID`.

`d_tcsetpgrp`
From *d_tcstpgrp.U*:

This variable conditionally defines the `HAS_TCSETPGRP` symbol, which indicates to the C program that the *tcsetpgrp()* routine is available to set foreground process group `ID`.

`d_telldir`
From *d_readdir.U*:

This variable conditionally defines `HAS_TELLDIR` if *telldir()* is available.

`d_time`
From *d_time.U*:

This variable conditionally defines the `HAS_TIME` symbol, which indicates that the *time()* routine exists. The *time()* routine is normaly provided on `UNIX` systems.

`d_times`
From *d_times.U*:

This variable conditionally defines the `HAS_TIMES` symbol, which indicates that the *times()* routine exists. The *times()* routine is normaly provided on `UNIX` systems. You may have to include <sys/times.h>.

`d_truncate`
From *d_truncate.U*:

This variable conditionally defines `HAS_TRUNCATE` if *truncate()* is available to truncate files.

`d_tzname`
From *d_tzname.U*:

This variable conditionally defines `HAS_TZNAME` if tzname[] is available to access timezone names.

`d_umask`
From *d_umask.U*:

This variable conditionally defines the `HAS_UMASK` symbol, which indicates to the C program that the *umask()* routine is available. to set and get the value of the file creation mask.

`d_uname`
From *d_gethname.U*:

This variable conditionally defines the `HAS_UNAME` symbol, which indicates to the C program that the *uname()* routine may be used to derive the host name.

`d_union_semun`
From *d_union_senum.U*:

This variable conditionally defines `HAS_UNION_SEMUN` if the union semun is defined by including <sys/sem.h>.

`d_vfork`
From *d_vfork.U*:

This variable conditionally defines the `HAS_VFORK` symbol, which indicates the *vfork()* routine is available.

`d_void_closedir`
From *d_closedir.U*:

This variable conditionally defines `VOID_CLOSEDIR` if *closedir()* does not return a value.

`d_voidsig`
From *d_voidsig.U*:

This variable conditionally defines `VOIDSIG` if this system declares "void (**signal* (...))()" in *signal.h*. The old way was to declare it as "int (**signal* (...))()".

`d_voidtty`
From *i_sysioctl.U*:

This variable conditionally defines `USE_IOCNOTTY` to indicate that the *ioctl()* call with `TIOCNOTTY` should be used to void tty association. Otherwise (on `USG` probably), it is enough to close the standard file decriptors and do a *setpgrp()*.

`d_volatile`
From *d_volatile.U*:

This variable conditionally defines the `HASVOLATILE` symbol, which indicates to the C program that this C compiler knows about the volatile declaration.

`d_vprintf`
From *d_vprintf.U*:

This variable conditionally defines the `HAS_VPRINTF` symbol, which indicates to the C program that the *vprintf()* routine is available to printf with a pointer to an argument list.

`d_wait4`
From *d_wait4.U*:

This variable conditionally defines the HAS_WAIT4 symbol, which indicates the *wait4()* routine is available.

`d_waitpid`
From *d_waitpid.U*:

This variable conditionally defines `HAS_WAITPID` if *waitpid()* is available to wait for child process.

`d_wcstombs`
From *d_wcstombs.U*:

This variable conditionally defines the `HAS_WCSTOMBS` symbol, which indicates to the C program that the *wcstombs()* routine is available to convert wide character strings to multibyte strings.

`d_wctomb`
From *d_wctomb.U*:

This variable conditionally defines the `HAS_WCTOMB` symbol, which indicates to the C program that the *wctomb()* routine is available to convert a wide character to a multibyte.

`d_xenix`
From *Guess.U*:

This variable conditionally defines the symbol `XENIX`, which alerts the C program that it runs under Xenix.

`date`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the date program. After Configure runs, the value is reset to a plain `date` and is not useful.

`db_hashtype`
From *i_db.U*:

This variable contains the type of the hash structure element in the <db.h> header file. In older versions of `DB`, it was int, while in newer ones it is u_int32_t.

`db_prefixtype`
From *i_db.U*:

This variable contains the type of the prefix structure element in the <db.h> header file. In older versions of `DB`, it was int, while in newer ones it is size_t.

`direntrytype`
From *i_dirent.U*:

This symbol is set to `struct direct` or `struct dirent` depending on whether dirent is available or not. You should use this pseudo type to portably declare your directory entries.

`dlext`
From *dlext.U*:

This variable contains the extension that is to be used for the dynamically loaded modules that perl generaties.

`dlsrc`
From *dlsrc.U*:

This variable contains the name of the dynamic loading file that will be used with the package.

`doublesize`
From *doublesize.U*:

This variable contains the value of the `DOUBLESIZE` symbol, which indicates to the C program how many bytes there are in a double.

`dynamic_ext`
From *Extensions.U*:

This variable holds a list of `XS` extension files we want to link dynamically into the package. It is used by Makefile.

**e**

`eagain`
From *nblock_io.U*:

This variable bears the symbolic errno code set by *read()* when no data is present on the file and non-blocking *I/O* was enabled (otherwise, *read()* blocks naturally).

`ebcdic`
From *ebcdic.U*:

This variable conditionally defines `EBCDIC` if this system uses `EBCDIC` encoding. Among other things, this means that the character ranges are not contiguous. See *trnl.U*

`echo`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the echo program. After Configure runs, the value is reset to a plain `echo` and is not useful.

`egrep`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the egrep program. After Configure runs, the value is reset to a plain `egrep` and is not useful.

`emacs`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`eunicefix`
From *Init.U*:

When running under Eunice this variable contains a command which will convert a shell script to the proper form of text file for it to be executable by the shell. On other systems it is a no-op.

`exe_ext`
From *Unix.U*:

This is an old synonym for _exe.

`expr`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the expr program. After Configure runs, the value is reset to a plain `expr` and is not useful.

`extensions`
From *Extensions.U*:

This variable holds a list of all extension files (both `XS` and non-xs linked into the package. It is propagated to *Config.pm* and is typically used to test whether a particular extesion is available.

**f**

`find`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the find program. After Configure runs, the value is reset to a plain `find` and is not useful.

`firstmakefile`
From *Unix.U*:

This variable defines the first file searched by make. On unix, it is makefile (then Makefile). On case-insensitive systems, it might be something else. This is only used to deal with convoluted make depend tricks.

`flex`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`fpostype`
From *fpostype.U*:

This variable defines Fpos_t to be something like fpost_t, long, uint, or whatever type is used to declare file positions in libc.

`freetype`
From *mallocsrc.U*:

This variable contains the return type of *free()*. It is usually void, but occasionally int.

`full_csh`
From *d_csh.U*:

This variable contains the full pathname to `csh`, whether or not the user has specified `portability`. This is only used in the compiled C program, and we assume that all systems which can share this executable will have the same full pathname to *csh.*

`full_sed`
From *Loc_sed.U*:

This variable contains the full pathname to `sed`, whether or not the user has specified `portability`. This is only used in the compiled C program, and we assume that all systems which can share this executable will have the same full pathname to *sed.*

**g**

`gccversion`
From *cc.U*:

If GNU cc (gcc) is used, this variable holds `1` or `2` to indicate whether the compiler is version 1 or 2. This is used in setting some of the default cflags. It is set to '' if not gcc.

`gidtype`
From *gidtype.U*:

This variable defines Gid_t to be something like gid_t, int, ushort, or whatever type is used to declare the return type of *getgid()*. Typically, it is the type of group ids in the kernel.

`grep`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the grep program. After Configure runs, the value is reset to a plain `grep` and is not useful.

`groupcat`
From *nis.U*:

This variable contains a command that produces the text of the */etc/group* file. This is normally ''cat */etc/group*'', but can be ''ypcat group'' when `NIS` is used.

`groupstype`
From *groupstype.U*:

This variable defines Groups_t to be something like gid_t, int, ushort, or whatever type is used for the second argument to *getgroups()* and *setgroups()*. Usually, this is the same as gidtype (gid_t), but sometimes it isn't.

`gzip`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the gzip program. After Configure runs, the value is reset to a plain `gzip` and is not useful.

**h**

`h_fcntl`
From *h_fcntl.U*:

This is variable gets set in various places to tell i_fcntl that <fcntl.h> should be included.

`h_sysfile`
From *h_sysfile.U*:

This is variable gets set in various places to tell i_sys_file that <sys/file.h> should be included.

`hint`
From *Oldconfig.U*:

Gives the type of hints used for previous answers. May be one of `default`, `recommended` or `previous`.

`hostcat`
From *nis.U*:

This variable contains a command that produces the text of the */etc/hosts* file. This is normally ''cat */etc/hosts*'', but can be ''ypcat hosts'' when `NIS` is used.

`huge`
From *models.U*:

This variable contains a flag which will tell the C compiler and loader to produce a program running with a huge memory model. If the huge model is not supported, contains the flag to produce large model programs. It is up to the Makefile to use this.

**i**

`i_arpainet`
From *i_arpainet.U*:

This variable conditionally defines the `I_ARPA_INET` symbol, and indicates whether a C program should include <arpa/inet.h>.

`i_bsdioctl`
From *i_sysioctl.U*:

This variable conditionally defines the `I_SYS_BSDIOCTL` symbol, which indicates to the C program that <sys/bsdioctl.h> exists and should be included.

`i_db`
From *i_db.U*:

This variable conditionally defines the `I_DB` symbol, and indicates whether a C program may include Berkeley's `DB` include file <db.h>.

`i_dbm`
From *i_dbm.U*:

This variable conditionally defines the `I_DBM` symbol, which indicates to the C program that <dbm.h> exists and should be included.

`i_dirent`
From *i_dirent.U*:

This variable conditionally defines `I_DIRENT`, which indicates to the C program that it should include <dirent.h>.

`i_dld`
From *i_dld.U*:

This variable conditionally defines the `I_DLD` symbol, which indicates to the C program that <dld.h> (`GNU` dynamic loading) exists and should be included.

`i_dlfcn`
From *i_dlfcn.U*:

This variable conditionally defines the `I_DLFCN` symbol, which indicates to the C program that <dlfcn.h> exists and should be included.

`i_fcntl`
From *i_fcntl.U*:

This variable controls the value of `I_FCNTL` (which tells the C program to include <fcntl.h>).

`i_float`
From *i_float.U*:

This variable conditionally defines the `I_FLOAT` symbol, and indicates whether a C program may include <float.h> to get symbols like `DBL_MAX` or `DBL_MIN`, *i.e*. machine dependent floating point values.

`i_gdbm`
From *i_gdbm.U*:

This variable conditionally defines the `I_GDBM` symbol, which indicates to the C program that <gdbm.h> exists and should be included.

`i_grp`
From *i_grp.U*:

This variable conditionally defines the `I_GRP` symbol, and indicates whether a C program should include <grp.h>.

`i_limits`
From *i_limits.U*:

This variable conditionally defines the `I_LIMITS` symbol, and indicates whether a C program may include <limits.h> to get symbols like `WORD_BIT` and friends.

`i_locale`
From *i_locale.U*:

This variable conditionally defines the `I_LOCALE` symbol, and indicates whether a C program should include <locale.h>.

`i_malloc`
From *i_malloc.U*:

This variable conditionally defines the `I_MALLOC` symbol, and indicates whether a C program should include <malloc.h>.

`i_math`
From *i_math.U*:

This variable conditionally defines the `I_MATH` symbol, and indicates whether a C program may include <math.h>.

`i_memory`
From *i_memory.U*:

This variable conditionally defines the `I_MEMORY` symbol, and indicates whether a C program should include <memory.h>.

`i_ndbm`
From *i_ndbm.U*:

This variable conditionally defines the `I_NDBM` symbol, which indicates to the C program that <ndbm.h> exists and should be included.

`i_netdb`
From *i_netdb.U*:

This variable conditionally defines the `I_NETDB` symbol, and indicates whether a C program should include <netdb.h>.

`i_neterrno`
From *i_neterrno.U*:

This variable conditionally defines the `I_NET_ERRNO` symbol, which indicates to the C program that <net/errno.h> exists and should be included.

`i_niin`
From *i_niin.U*:

This variable conditionally defines `I_NETINET_IN`, which indicates to the C program that it should include <netinet/in.h>. Otherwise, you may try <sys/in.h>.

`i_pwd`
From *i_pwd.U*:

This variable conditionally defines `I_PWD`, which indicates to the C program that it should include <pwd.h>.

`i_rpcsvcdbm`
From *i_dbm.U*:

This variable conditionally defines the `I_RPCSVC_DBM` symbol, which indicates to the C program that <rpcsvc/dbm.h> exists and should be included. Some System V systems might need this instead of <dbm.h>.

`i_sfio`
From *i_sfio.U*:

This variable conditionally defines the `I_SFIO` symbol, and indicates whether a C program should include <sfio.h>.

`i_sgtty`
From *i_termio.U*:

This variable conditionally defines the `I_SGTTY` symbol, which indicates to the C program that it should include <sgtty.h> rather than <termio.h>.

`i_stdarg`
From *i_varhdr.U*:

This variable conditionally defines the `I_STDARG` symbol, which indicates to the C program that <stdarg.h> exists and should be included.

`i_stddef`
From *i_stddef.U*:

This variable conditionally defines the `I_STDDEF` symbol, which indicates to the C program that <stddef.h> exists and should be included.

`i_stdlib`
From *i_stdlib.U*:

This variable conditionally defines the `I_STDLIB` symbol, which indicates to the C program that <stdlib.h> exists and should be included.

`i_string`
From *i_string.U*:

This variable conditionally defines the `I_STRING` symbol, which indicates that <string.h> should be included rather than <strings.h>.

`i_sysdir`
From *i_sysdir.U*:

This variable conditionally defines the `I_SYS_DIR` symbol, and indicates whether a C program should include <sys/dir.h>.

`i_sysfile`
From *i_sysfile.U*:

This variable conditionally defines the `I_SYS_FILE` symbol, and indicates whether a C program should include <sys/file.h> to get `R_OK` and friends.

`i_sysfilio`
From *i_sysioctl.U*:

This variable conditionally defines the `I_SYS_FILIO` symbol, which indicates to the C program that <sys/filio.h> exists and should be included in preference to <sys/ioctl.h>.

`i_sysin`
From *i_niin.U*:

This variable conditionally defines `I_SYS_IN`, which indicates to the C program that it should include <sys/in.h> instead of <netinet/in.h>.

`i_sysioctl`
From *i_sysioctl.U*:

This variable conditionally defines the `I_SYS_IOCTL` symbol, which indicates to the C program that <sys/ioctl.h> exists and should be included.

`i_sysndir`
From *i_sysndir.U*:

This variable conditionally defines the `I_SYS_NDIR` symbol, and indicates whether a C program should include <sys/ndir.h>.

`i_sysparam`
From *i_sysparam.U*:

This variable conditionally defines the `I_SYS_PARAM` symbol, and indicates whether a C program should include <sys/param.h>.

`i_sysresrc`
From *i_sysresrc.U*:

This variable conditionally defines the `I_SYS_RESOURCE` symbol, and indicates whether a C program should include <sys/resource.h>.

`i_sysselct`
From *i_sysselct.U*:

This variable conditionally defines `I_SYS_SELECT`, which indicates to the C program that it should include <sys/select.h> in order to get the definition of struct timeval.

`i_syssockio`
From *i_sysioctl.U*:

This variable conditionally defines `I_SYS_SOCKIO` to indicate to the C program that socket ioctl codes may be found in <sys/sockio.h> instead of <sys/ioctl.h>.

`i_sysstat`
From *i_sysstat.U*:

This variable conditionally defines the `I_SYS_STAT` symbol, and indicates whether a C program should include <sys/stat.h>.

`i_systime`
From *i_time.U*:

This variable conditionally defines `I_SYS_TIME`, which indicates to the C program that it should include <sys/time.h>.

`i_systimek`
From *i_time.U*:

This variable conditionally defines `I_SYS_TIME_KERNEL`, which indicates to the C program that it should include <sys/time.h> with `KERNEL` defined.

`i_systimes`
From *i_systimes.U*:

This variable conditionally defines the `I_SYS_TIMES` symbol, and indicates whether a C program should include <sys/times.h>.

`i_systypes`
From *i_systypes.U*:

This variable conditionally defines the `I_SYS_TYPES` symbol, and indicates whether a C program should include <sys/types.h>.

`i_sysun`
From *i_sysun.U*:

This variable conditionally defines `I_SYS_UN`, which indicates to the C program that it should include <sys/un.h> to get `UNIX` domain socket definitions.

`i_syswait`
From *i_syswait.U*:

This variable conditionally defines `I_SYS_WAIT`, which indicates to the C program that it should include <sys/wait.h>.

`i_termio`
From *i_termio.U*:

This variable conditionally defines the `I_TERMIO` symbol, which indicates to the C program that it should include <termio.h> rather than <sgtty.h>.

`i_termios`
From *i_termio.U*:

This variable conditionally defines the `I_TERMIOS` symbol, which indicates to the C program that the `POSIX` <termios.h> file is to be included.

`i_time`
From *i_time.U*:

This variable conditionally defines `I_TIME`, which indicates to the C program that it should include <time.h>.

`i_unistd`
From *i_unistd.U*:

This variable conditionally defines the `I_UNISTD` symbol, and indicates whether a C program should include <unistd.h>.

`i_utime`
From *i_utime.U*:

This variable conditionally defines the `I_UTIME` symbol, and indicates whether a C program should include <utime.h>.

`i_values`
From *i_values.U*:

This variable conditionally defines the `I_VALUES` symbol, and indicates whether a C program may include <values.h> to get symbols like `MAXLONG` and friends.

`i_varargs`
From *i_varhdr.U*:

This variable conditionally defines `I_VARARGS`, which indicates to the C program that it should include <varargs.h>.

`i_varhdr`
From *i_varhdr.U*:

Contains the name of the header to be included to get va_dcl definition. Typically one of *varargs.h* or *stdarg.h*.

`i_vfork`
From *i_vfork.U*:

This variable conditionally defines the `I_VFORK` symbol, and indicates whether a C program should include *vfork.h*.

`incpath`
From *usrinc.U*:

This variable must preceed the normal include path to get hte right one, as in *$incpath/usr/include* or *$incpath/usr/lib*. Value can be "" or */bsd43* on mips.

`inews`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`installarchlib`
From *archlib.U*:

This variable is really the same as archlibexp but may differ on those systems using `AFS`. For extra portability, only this variable should be used in makefiles.

`installbin`
From *bin.U*:

This variable is the same as binexp unless AFS is running in which case the user is explicitely prompted for it. This variable should always be used in your makefiles for maximum portability.

`installman1dir`
From *man1dir.U*:

This variable is really the same as man1direxp, unless you are using AFS in which case it points to the *read/write* location whereas man1direxp only points to the read-only access location. For extra portability, you should only use this variable within your makefiles.

`installman3dir`
From *man3dir.U*:

This variable is really the same as man3direxp, unless you are using AFS in which case it points to the *read/write* location whereas man3direxp only points to the read-only access location. For extra portability, you should only use this variable within your makefiles.

`installprivlib`
From *privlib.U*:

This variable is really the same as privlibexp but may differ on those systems using AFS. For extra portability, only this variable should be used in makefiles.

`installscript`
From *scriptdir.U*:

This variable is usually the same as scriptdirexp, unless you are on a system running AFS, in which case they may differ slightly. You should always use this variable within your makefiles for portability.

`installsitearch`
From *sitearch.U*:

This variable is really the same as sitearchexp but may differ on those systems using AFS. For extra portability, only this variable should be used in makefiles.

`installsitelib`
From *sitelib.U*:

This variable is really the same as sitelibexp but may differ on those systems using AFS. For extra portability, only this variable should be used in makefiles.

`intsize`
From *intsize.U*:

This variable contains the value of the INTSIZE symbol, which indicates to the C program how many bytes there are in an int.

**k**

`known_extensions`
From *Extensions.U*:

This variable holds a list of all XS extensions included in the package.

`ksh` From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

**l**

`large`
From *models.U*:

This variable contains a flag which will tell the C compiler and loader to produce a program running with a large memory model. It is up to the Makefile to use this.

`ld` From *dlsrc.U*:

This variable indicates the program to be used to link libraries for dynamic loading. On some systems, it is `ld`. On `ELF` systems, it should be `$cc`. Mostly, we'll try to respect the hint file setting.

`lddlflags`
From *dlsrc.U*:

This variable contains any special flags that might need to be passed to `$ld` to create a shared library suitable for dynamic loading. It is up to the makefile to use it. For hpux, it should be `-b`. For sunos 4.1, it is empty.

`ldflags`
From *ccflags.U*:

This variable contains any additional C loader flags desired by the user. It is up to the Makefile to use this.

`less`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the less program. After Configure runs, the value is reset to a plain `less` and is not useful.

`lib_ext`
From *Unix.U*:

This is an old synonym for _a.

`libc`
From *libc.U*:

This variable contains the location of the C library.

`libperl`
From *libperl.U*:

The perl executable is obtained by linking *perlmain.c* with libperl, any static extensions (usually just DynaLoader), and any other libraries needed on this system. libperl is usually *libperl.a*, but can also be *libperl.so.xxx* if the user wishes to build a perl executable with a shared library.

`libpth`
From *libpth.U*:

This variable holds the general path used to find libraries. It is intended to be used by other units.

`libs`
From *libs.U*:

This variable holds the additional libraries we want to use. It is up to the Makefile to deal with it.

`libswanted`
From *Myinit.U*:

This variable holds a list of all the libraries we want to search. The order is chosen to pick up the c library ahead of ucb or bsd libraries for SVR4.

`line`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the line program. After Configure runs, the value is reset to a plain `line` and is not useful.

`lint`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`lkflags`
From *ccflags.U*:

This variable contains any additional C partial linker flags desired by the user. It is up to the Makefile to use this.

`ln` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the ln program. After Configure runs, the value is reset to a plain `ln` and is not useful.

`lns` From *lns.U*:

This variable holds the name of the command to make symbolic links (if they are supported). It can be used in the Makefile. It is either `ln -s` or `ln`

`locincpth`
From *ccflags.U*:

This variable contains a list of additional directories to be searched by the compiler. The appropriate `-I` directives will be added to ccflags. This is intended to simplify setting local directories from the Configure command line. It's not much, but it parallels the loclibpth stuff in *libpth.U*.

`loclibpth`
From *libpth.U*:

This variable holds the paths used to find local libraries. It is prepended to libpth, and is intended to be easily set from the command line.

`longdblsize`
From *d_longdbl.U*:

This variable contains the value of the `LONG_DOUBLESIZE` symbol, which indicates to the C program how many bytes there are in a long double, if this system supports long doubles.

`longlongsize`
From *d_longlong.U*:

This variable contains the value of the `LONGLONGSIZE` symbol, which indicates to the C program how many bytes there are in a long long, if this system supports long long.

`longsize`
From *intsize.U*:

This variable contains the value of the `LONGSIZE` symbol, which indicates to the C program how many bytes there are in a long.

`lp` From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`lpr` From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`ls` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the ls program. After Configure runs, the value is reset to a plain `ls` and is not useful.

`lseektype`
From *lseektype.U*:

This variable defines lseektype to be something like off_t, long, or whatever type is used to declare lseek offset's type in the kernel (which also appears to be lseek's return type).

**m**

`mail`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`mailx`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`make`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the make program. After Configure runs, the value is reset to a plain `make` and is not useful.

`make_set_make`
From *make.U*:

Some versions of `make` set the variable `MAKE`. Others do not. This variable contains the string to be included in *Makefile.SH* so that `MAKE` is set if needed, and not if not needed. Possible values are: make_set_make=`#` # If your make program handles this for you, make_set_make=`MAKE=$make` # if it doesn't. I used a comment character so that we can distinguish a `set` value (from a previous *config.sh* or Configure `-D` option) from an uncomputed value.

`mallocobj`
From *mallocsrc.U*:

This variable contains the name of the *malloc.o* that this package generates, if that *malloc.o* is preferred over the system malloc. Otherwise the value is null. This variable is intended for generating Makefiles. See mallocsrc.

`mallocsrc`
From *mallocsrc.U*:

This variable contains the name of the *malloc.c* that comes with the package, if that *malloc.c* is preferred over the system malloc. Otherwise the value is null. This variable is intended for generating Makefiles.

`malloctype`
From *mallocsrc.U*:

This variable contains the kind of ptr returned by malloc and realloc.

`man1dir`
From *man1dir.U*:

This variable contains the name of the directory in which manual source pages are to be put. It is the responsibility of the *Makefile.SH* to get the value of this into the proper command. You must be prepared to do the *~name* expansion yourself.

`man1direxp`

From *man1dir.U*:

This variable is the same as the man1dir variable, but is filename expanded at configuration time, for convenient use in makefiles.

`man1ext`

From *man1dir.U*:

This variable contains the extension that the manual page should have: one of `n`, `l`, or `1`. The Makefile must supply the .. See man1dir.

`man3dir`

From *man3dir.U*:

This variable contains the name of the directory in which manual source pages are to be put. It is the responsibility of the *Makefile.SH* to get the value of this into the proper command. You must be prepared to do the *~name* expansion yourself.

`man3direxp`

From *man3dir.U*:

This variable is the same as the man3dir variable, but is filename expanded at configuration time, for convenient use in makefiles.

`man3ext`

From *man3dir.U*:

This variable contains the extension that the manual page should have: one of `n`, `l`, or `3`. The Makefile must supply the .. See man3dir.

`medium`

From *models.U*:

This variable contains a flag which will tell the C compiler and loader to produce a program running with a medium memory model. If the medium model is not supported, contains the flag to produce large model programs. It is up to the Makefile to use this.

`mips_type`

From *usrinc.U*:

This variable holds the environment type for the mips system. Possible values are "BSD 4.3" and "System V".

`mkdir`

From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the mkdir program. After Configure runs, the value is reset to a plain `mkdir` and is not useful.

`models`

From *models.U*:

This variable contains the list of memory models supported by this system. Possible component values are none, split, unsplit, small, medium, large, and huge. The component values are space separated.

`modetype`

From *modetype.U*:

This variable defines modetype to be something like mode_t, int, unsigned short, or whatever type is used to declare file modes for system calls.

`more`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the more program. After Configure runs, the value is reset to a plain `more` and is not useful.

`mv` From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`myarchname`
From *archname.U*:

This variable holds the architecture name computed by Configure in a previous run. It is not intended to be perused by any user and should never be set in a hint file.

`mydomain`
From *myhostname.U*:

This variable contains the eventual value of the `MYDOMAIN` symbol, which is the domain of the host the program is going to run on. The domain must be appended to myhostname to form a complete host name. The dot comes with mydomain, and need not be supplied by the program.

`myhostname`
From *myhostname.U*:

This variable contains the eventual value of the `MYHOSTNAME` symbol, which is the name of the host the program is going to run on. The domain is not kept with hostname, but must be gotten from mydomain. The dot comes with mydomain, and need not be supplied by the program.

`myuname`
From *Oldconfig.U*:

The output of `uname -a` if available, otherwise the hostname. On Xenix, pseudo variables assignments in the output are stripped, thank you. The whole thing is then lower-cased.

**n**

`n` From *n.U*:

This variable contains the `-n` flag if that is what causes the echo command to suppress newline. Otherwise it is null. Correct usage is

```
$echo $n "prompt for a question: $c".
```

`netdb_hlen_type`
From *netdbtype.U*:

This variable holds the type used for the 2nd argument to *gethostbyaddr()*. Usually, this is int or size_t or unsigned. This is only useful if you have *gethostbyaddr()*, naturally.

`netdb_host_type`
From *netdbtype.U*:

This variable holds the type used for the 1st argument to *gethostbyaddr()*. Usually, this is char * or void *, possibly with or without a const prefix. This is only useful if you have *gethostbyaddr()*, naturally.

`netdb_name_type`
From *netdbtype.U*:

This variable holds the type used for the argument to *gethostbyname()*. Usually, this is char * or const char *. This is only useful if you have *gethostbyname()*, naturally.

`netdb_net_type`
From *netdbtype.U*:

This variable holds the type used for the 1st argument to *getnetbyaddr()*. Usually, this is int or long. This is only useful if you have *getnetbyaddr()*, naturally.

`nm`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the nm program. After Configure runs, the value is reset to a plain `nm` and is not useful.

`nm_opt`
From *usenm.U*:

This variable holds the options that may be necessary for nm.

`nm_so_opt`
From *usenm.U*:

This variable holds the options that may be necessary for nm to work on a shared library but that can not be used on an archive library. Currently, this is only used by Linux, where nm --dynamic is *required* to get symbols from an `ELF` library which has been stripped, but nm --dynamic is *fatal* on an archive library. Maybe Linux should just always set usenm=false.

`nonxs_ext`
From *Extensions.U*:

This variable holds a list of all non-xs extensions included in the package. All of them will be built.

`nroff`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the nroff program. After Configure runs, the value is reset to a plain `nroff` and is not useful.

**o**

`o_nonblock`
From *nblock_io.U*:

This variable bears the symbol value to be used during *open()* or *fcntl()* to turn on non-blocking *I/O* for a file descriptor. If you wish to switch between blocking and non-blocking, you may try *ioctl*(`FIOSNBIO`) instead, but that is only supported by some devices.

`obj_ext`
From *Unix.U*:

This is an old synonym for _o.

`optimize`
From *ccflags.U*:

This variable contains any *optimizer/debugger* flag that should be used. It is up to the Makefile to use it.

`orderlib`
From *orderlib.U*:

This variable is `true` if the components of libraries must be ordered (with ‘lorder $* | tsort‘) before placing them in an archive. Set to `false` if ranlib or ar can generate random libraries.

`osname`
From *Oldconfig.U*:

This variable contains the operating system name (e.g. sunos, solaris, hpux, *etc.*). It can be useful later on for setting defaults. Any spaces are replaced with underscores. It is set to a null string if we can't figure it out.

`osvers`

From *Oldconfig.U*:

This variable contains the operating system version (e.g. 4.1.3, 5.2, *etc.*). It is primarily used for helping select an appropriate hints file, but might be useful elsewhere for setting defaults. It is set to '' if we can't figure it out. We try to be flexible about how much of the version number to keep, e.g. if 4.1.1, 4.1.2, and 4.1.3 are essentially the same for this package, hints files might just be *os_4.0* or *os_4.1*, *etc.*, not keeping separate files for each little release.

**p**

`package`

From *package.U*:

This variable contains the name of the package being constructed. It is primarily intended for the use of later Configure units.

`pager`

From *pager.U*:

This variable contains the name of the preferred pager on the system. Usual values are (the full pathnames of) more, less, pg, or cat.

`passcat`

From *nis.U*:

This variable contains a command that produces the text of the */etc/passwd* file. This is normally "cat */etc/passwd*", but can be "ypcat passwd" when `NIS` is used.

`patchlevel`

From *patchlevel.U*:

The patchlevel level of this package. The value of patchlevel comes from the *patchlevel.h* file.

`path_sep`

From *Unix.U*:

This is an old synonym for p_ in *Head.U*, the character used to separate elements in the command shell search `PATH`.

`perl`

From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the perl program. After Configure runs, the value is reset to a plain `perl` and is not useful.

`perladmin`

From *perladmin.U*:

Electronic mail address of the perl5 administrator.

`perlpath`

From *perlpath.U*:

This variable contains the eventual value of the `PERLPATH` symbol, which contains the name of the perl interpreter to be used in shell scripts and in the "eval `exec`" idiom.

`pg`

From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the pg program. After Configure runs, the value is reset to a plain `pg` and is not useful.

`phostname`
From *myhostname.U*:

This variable contains the eventual value of the PHOSTNAME symbol, which is a command that can be fed to *popen()* to get the host name. The program should probably not presume that the domain is or isn't there already.

`pidtype`
From *pidtype.U*:

This variable defines PIDTYPE to be something like pid_t, int, ushort, or whatever type is used to declare process ids in the kernel.

`plibpth`
From *libpth.U*:

Holds the private path used by Configure to find out the libraries. Its value is prepend to libpth. This variable takes care of special machines, like the mips. Usually, it should be empty.

`pmake`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`pr` From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`prefix`
From *prefix.U*:

This variable holds the name of the directory below which the user will install the package. Usually, this is */usr/local*, and executables go in */usr/local/bin*, library stuff in */usr/local/lib*, man pages in */usr/local/man*, etc. It is only used to set defaults for things in *bin.U*, *mansrc.U*, *privlib.U*, or *scriptdir.U*.

`prefixexp`
From *prefix.U*:

This variable holds the full absolute path of the directory below which the user will install the package. Derived from prefix.

`privlib`
From *privlib.U*:

This variable contains the eventual value of the PRIVLIB symbol, which is the name of the private library for this package. It may have a ~ on the front. It is up to the makefile to eventually create this directory while performing installation (with ~ substitution).

`privlibexp`
From *privlib.U*:

This variable is the *~name* expanded version of privlib, so that you may use it directly in Makefiles or shell scripts.

`prototype`
From *prototype.U*:

This variable holds the eventual value of CAN_PROTOTYPE, which indicates the C compiler can handle funciton prototypes.

`ptrsize`
From *ptrsize.U*:

This variable contains the value of the PTRSIZE symbol, which indicates to the C program how many bytes there are in a pointer.

**r**

`randbits`
From *randbits.U*:

This variable contains the eventual value of the `RANDBITS` symbol, which indicates to the C program how many bits of random number the *rand()* function produces.

`ranlib`
From *orderlib.U*:

This variable is set to the pathname of the ranlib program, if it is needed to generate random libraries. Set to **:** if ar can generate random libraries or if random libraries are not supported

`rd_nodata`
From *nblock_io.U*:

This variable holds the return code from *read()* when no data is present. It should be −1, but some systems return 0 when `O_NDELAY` is used, which is a shame because you cannot make the difference between no data and an *EOF.*. Sigh!

`rm` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the rm program. After Configure runs, the value is reset to a plain `rm` and is not useful.

`rmail`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`runnm`
From *usenm.U*:

This variable contains `true` or `false` depending whether the nm extraction should be performed or not, according to the value of usenm and the flags on the Configure command line.

**s**

`scriptdir`
From *scriptdir.U*:

This variable holds the name of the directory in which the user wants to put publicly scripts for the package in question. It is either the same directory as for binaries, or a special one that can be mounted across different architectures, like */usr/share*. Programs must be prepared to deal with *~name* expansion.

`scriptdirexp`
From *scriptdir.U*:

This variable is the same as scriptdir, but is filename expanded at configuration time, for programs not wanting to bother with it.

`sed` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the sed program. After Configure runs, the value is reset to a plain `sed` and is not useful.

`selecttype`
From *selecttype.U*:

This variable holds the type used for the 2nd, 3rd, and 4th arguments to select. Usually, this is `fd_set *`, if `HAS_FD_SET` is defined, and `int *` otherwise. This is only useful if you have *select()*, naturally.

`sendmail`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the sendmail program. After Configure runs, the value is reset to a plain `sendmail` and is not useful.

`sh` From *sh.U*:

This variable contains the full pathname of the shell used on this system to execute Bourne shell scripts. Usually, this will be */bin/sh*, though it's possible that some systems will have */bin/ksh*, */bin/pdksh*, */bin/ash*, */bin/bash*, or even something such as D:*/bin/sh.exe*. This unit comes before *Options.U*, so you can't set sh with a `-D` option, though you can override this (and startsh) with `-O -Dsh=`*/bin/whatever* -Dstartsh=whatever

`shar`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`sharpbang`
From *spitshell.U*:

This variable contains the string #! if this system supports that construct.

`shmattype`
From *d_shmat.U*:

This symbol contains the type of pointer returned by *shmat()*. It can be `void *` or `char *`.

`shortsize`
From *intsize.U*:

This variable contains the value of the `SHORTSIZE` symbol which indicates to the C program how many bytes there are in a short.

`shrpenv`
From *libperl.U*:

If the user builds a shared *libperl.so*, then we need to tell the `perl` executable where it will be able to find the installed *libperl.so*. One way to do this on some systems is to set the environment variable `LD_RUN_PATH` to the directory that will be the final location of the shared *libperl.so*. The makefile can use this with something like

```
        $shrpenv $(C<CC>) -o perl F<perlmain.o> $libperl $libs

        Typical values are

        shrpenv="env C<LD_RUN_PATH>=$F<archlibexp/C<CORE>>"

        or

        shrpenv=''

        See the main perl F<Makefile.SH> for actual working usage.
Alternatively, we might be able to use a command line option such
as -R $F<archlibexp/C<CORE>> (Solaris, NetBSD) or -Wl,-rpath
$F<archlibexp/C<CORE>> (Linux).
```

`shsharp`
From *spitshell.U*:

This variable tells further Configure units whether your sh can handle # comments.

`sig_name`
From *sig_name.U*:

This variable holds the signal names, space separated. The leading `SIG` in signal name is removed. A `ZERO` is prepended to the list. This is currently not used.

`sig_name_init`
From *sig_name.U*:

This variable holds the signal names, enclosed in double quotes and separated by commas, suitable for use in the `SIG_NAME` definition below. A `ZERO` is prepended to the list, and the list is terminated with a plain 0. The leading `SIG` in signal names is removed. See sig_num.

`sig_num`
From *sig_name.U*:

This variable holds the signal numbers, comma separated. A 0 is prepended to the list (corresponding to the fake `SIGZERO`), and the list is terminated with a 0. Those numbers correspond to the value of the signal listed in the same place within the sig_name list.

`signal_t`
From *d_voidsig.U*:

This variable holds the type of the signal handler (void or int).

`sitearch`
From *sitearch.U*:

This variable contains the eventual value of the `SITEARCH` symbol, which is the name of the private library for this package. It may have a ˜ on the front. It is up to the makefile to eventually create this directory while performing installation (with ˜ substitution).

`sitearchexp`
From *sitearch.U*:

This variable is the *˜name* expanded version of sitearch, so that you may use it directly in Makefiles or shell scripts.

`sitelib`
From *sitelib.U*:

This variable contains the eventual value of the `SITELIB` symbol, which is the name of the private library for this package. It may have a ˜ on the front. It is up to the makefile to eventually create this directory while performing installation (with ˜ substitution).

`sitelibexp`
From *sitelib.U*:

This variable is the *˜name* expanded version of sitelib, so that you may use it directly in Makefiles or shell scripts.

`sizetype`
From *sizetype.U*:

This variable defines sizetype to be something like size_t, unsigned long, or whatever type is used to declare length parameters for string functions.

`sleep`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`smail`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`small`
From *models.U*:

This variable contains a flag which will tell the C compiler and loader to produce a program running with a small memory model. It is up to the Makefile to use this.

`so` From *so.U*:

This variable holds the extension used to identify shared libraries (also known as shared objects) on the system. Usually set to `so`.

`sockethdr`
From *d_socket.U*:

This variable has any cpp `-I` flags needed for socket support.

`socketlib`
From *d_socket.U*:

This variable has the names of any libraries needed for socket support.

`sort`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the sort program. After Configure runs, the value is reset to a plain `sort` and is not useful.

`spackage`
From *package.U*:

This variable contains the name of the package being constructed, with the first letter uppercased, *i.e*. suitable for starting sentences.

`spitshell`
From *spitshell.U*:

This variable contains the command necessary to spit out a runnable shell on this system. It is either cat or a grep `-v` for # comments.

`split`
From *models.U*:

This variable contains a flag which will tell the C compiler and loader to produce a program that will run in separate I and D space, for those machines that support separation of instruction and data space. It is up to the Makefile to use this.

`src` From *src.U*:

This variable holds the path to the package source. It is up to the Makefile to use this variable and set `VPATH` accordingly to find the sources remotely.

`ssizetype`
From *ssizetype.U*:

This variable defines ssizetype to be something like ssize_t, long or int. It is used by functions that return a count of bytes or an error condition. It must be a signed type. We will pick a type such that *sizeof*(SSize_t) == *sizeof*(Size_t).

`startperl`
From *startperl.U*:

This variable contains the string to put on the front of a perl script to make sure (hopefully) that it runs with perl and not some shell. Of course, that leading line must be followed by the classical perl idiom:

```
eval 'exec perl -S $0 ${1+C<$@>}'
```

```
if $running_under_some_shell;
```

`to guarantee perl startup should the shell execute the script. Note that this magic incatation is not understood by csh.`

`startsh`
From *startsh.U*:

This variable contains the string to put on the front of a shell script to make sure (hopefully) that it runs with sh and not some other shell.

`static_ext`
From *Extensions.U*:

This variable holds a list of `XS` extension files we want to link statically into the package. It is used by Makefile.

`stdchar`
From *stdchar.U*:

This variable conditionally defines `STDCHAR` to be the type of char used in *stdio.h*. It has the values ''unsigned char'' or `char`.

`stdio_base`
From *d_stdstdio.U*:

This variable defines how, given a `FILE` pointer, fp, to access the _base field (or equivalent) of *stdio.h*'s `FILE` structure. This will be used to define the macro *FILE_base* (fp).

`stdio_bufsiz`
From *d_stdstdio.U*:

This variable defines how, given a `FILE` pointer, fp, to determine the number of bytes store in the *I/O* buffer pointer to by the _base field (or equivalent) of *stdio.h*'s `FILE` structure. This will be used to define the macro *FILE_bufsiz* (fp).

`stdio_cnt`
From *d_stdstdio.U*:

This variable defines how, given a `FILE` pointer, fp, to access the _cnt field (or equivalent) of *stdio.h*'s `FILE` structure. This will be used to define the macro *FILE_cnt* (fp).

`stdio_filbuf`
From *d_stdstdio.U*:

This variable defines how, given a `FILE` pointer, fp, to tell stdio to refill it's internal buffers (?). This will be used to define the macro *FILE_filbuf* (fp).

`stdio_ptr`
From *d_stdstdio.U*:

This variable defines how, given a `FILE` pointer, fp, to access the _ptr field (or equivalent) of *stdio.h*'s `FILE` structure. This will be used to define the macro *FILE_ptr* (fp).

`strings`
From *i_string.U*:

This variable holds the full path of the string header that will be used. Typically */usr/include/string.h* or */usr/include/strings.h*.

`submit`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`subversion`
From *patchlevel.U*:

The subversion level of this package. The value of subversion comes from the *patchlevel.h* file. This is unique to perl.

`sysman`
From *sysman.U*:

This variable holds the place where the manual is located on this system. It is not the place where the user wants to put his manual pages. Rather it is the place where Configure may look to find manual for unix commands (section 1 of the manual usually). See mansrc.

**t**

`tail`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`tar` From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`tbl` From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`tee` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the tee program. After Configure runs, the value is reset to a plain `tee` and is not useful.

`test`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the test program. After Configure runs, the value is reset to a plain `test` and is not useful.

`timeincl`
From *i_time.U*:

This variable holds the full path of the included time *header*(s).

`timetype`
From *d_time.U*:

This variable holds the type returned by *time()*. It can be long, or time_t on BSD sites (in which case <sys/types.h> should be included). Anyway, the type Time_t should be used.

`touch`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the touch program. After Configure runs, the value is reset to a plain `touch` and is not useful.

`tr` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the tr program. After Configure runs, the value is reset to a plain `tr` and is not useful.

`trnl`
From *trnl.U*:

This variable contains the value to be passed to the *tr*(1) command to transliterate a newline. Typical values are `\012` and `\n`. This is needed for EBCDIC systems where newline is not necessarily `\012`.

`troff`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

**u**

`uidtype`
From *uidtype.U*:

This variable defines Uid_t to be something like uid_t, int, ushort, or whatever type is used to declare user ids in the kernel.

`uname`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the uname program. After Configure runs, the value is reset to a plain `uname` and is not useful.

`uniq`
From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the uniq program. After Configure runs, the value is reset to a plain `uniq` and is not useful.

`usedl`
From *dlsrc.U*:

This variable indicates if the the system supports dynamic loading of some sort. See also dlsrc and dlobj.

`usemymalloc`
From *mallocsrc.U*:

This variable contains y if the malloc that comes with this package is desired over the system's version of malloc. People often include special versions of malloc for effiency, but such versions are often less portable. See also mallocsrc and mallocobj. If this is `y`, then −lmalloc is removed from `$libs`.

`usenm`
From *usenm.U*:

This variable contains `true` or `false` depending whether the nm extraction is wanted or not.

`useopcode`
From *Extensions.U*:

This variable holds either `true` or `false` to indicate whether the Opcode extension should be used. The sole use for this currently is to allow an easy mechanism for users to skip the Opcode extension from the Configure command line.

`useperlio`
From *useperlio.U*:

This variable conditionally defines the `USE_PERLIO` symbol, and indicates that the PerlIO abstraction should be used throughout.

`useposix`
From *Extensions.U*:

This variable holds either `true` or `false` to indicate whether the `POSIX` extension should be used. The sole use for this currently is to allow an easy mechanism for hints files to indicate that `POSIX` will not compile on a particular system.

`usesfio`
From *d_sfio.U*:

This variable is set to true when the user agrees to use sfio. It is set to false when sfio is not available or when the user explicitely requests not to use sfio. It is here primarily so that command-line settings can override the auto-detection of d_sfio without running into a ''WHOA THERE''.

`useshrplib`
From *libperl.U*:

This variable is set to `yes` if the user wishes to build a shared libperl, and `no` otherwise.

`usethreads`
From *usethreads.U*:

This variable conditionally defines the `USE_THREADS` symbol, and indicates that Perl should be built to use threads.

`usevfork`
From *d_vfork.U*:

This variable is set to true when the user accepts to use vfork. It is set to false when no vfork is available or when the user explicitely requests not to use vfork.

`usrinc`
From *usrinc.U*:

This variable holds the path of the include files, which is usually */usr/include*. It is mainly used by other Configure units.

`uuname`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

**v**

`version`
From *patchlevel.U*:

The full version number of this package. This combines baserev, patchlevel, and subversion to get the full version number, including any possible subversions. Care is taken to use the C locale in order to get something like 5.004 instead of 5,004. This is unique to perl.

`vi` From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`voidflags`
From *voidflags.U*:

This variable contains the eventual value of the `VOIDFLAGS` symbol, which indicates how much support of the void type is given by this compiler. See `VOIDFLAGS` for more info.

**z**

`zcat`
From *Loc.U*:

This variable is defined but not used by Configure. The value is a plain '' and is not useful.

`zip` From *Loc.U*:

This variable is be used internally by Configure to determine the full pathname (if any) of the zip program. After Configure runs, the value is reset to a plain `zip` and is not useful.

**NOTE**

This module contains a good example of how to use tie to implement a cache and an example of how to make a tied variable readonly to those outside of it.

**NAME**

CPAN – query, download and build perl modules from CPAN sites

**SYNOPSIS**

Interactive mode:

```
perl -MCPAN -e shell;
```

Batch mode:

```
use CPAN;

autobundle, clean, install, make, recompile, test
```

**DESCRIPTION**

The CPAN module is designed to automate the make and install of perl modules and extensions. It includes some searching capabilities and knows how to use Net::FTP or LWP (or lynx or an external ftp client) to fetch the raw data from the net.

Modules are fetched from one or more of the mirrored CPAN (Comprehensive Perl Archive Network) sites and unpacked in a dedicated directory.

The CPAN module also supports the concept of named and versioned 'bundles' of modules. Bundles simplify the handling of sets of related modules. See BUNDLES below.

The package contains a session manager and a cache manager. There is no status retained between sessions. The session manager keeps track of what has been fetched, built and installed in the current session. The cache manager keeps track of the disk space occupied by the make processes and deletes excess space according to a simple FIFO mechanism.

All methods provided are accessible in a programmer style and in an interactive shell style.

### Interactive Mode

The interactive mode is entered by running

```
perl -MCPAN -e shell
```

which puts you into a readline interface. You will have the most fun if you install Term::ReadKey and Term::ReadLine to enjoy both history and command completion.

Once you are on the command line, type 'h' and the rest should be self-explanatory.

The most common uses of the interactive modes are

Searching for authors, bundles, distribution files and modules

There are corresponding one-letter commands `a`, `b`, `d`, and `m` for each of the four categories and another, `i` for any of the mentioned four. Each of the four entities is implemented as a class with slightly differing methods for displaying an object.

Arguments you pass to these commands are either strings exactly matching the identification string of an object or regular expressions that are then matched case-insensitively against various attributes of the objects. The parser recognizes a regular expression only if you enclose it between two slashes.

The principle is that the number of found objects influences how an item is displayed. If the search finds one item, the result is displayed as object->as_string, but if we find more than one, we display each as object->as_glimpse. E.g.

```
cpan> a ANDK
Author id = ANDK
    EMAIL        a.koenig@franz.ww.TU-Berlin.DE
    FULLNAME     Andreas König
```

```
cpan> a /andk/
Author id = ANDK
    EMAIL        a.koenig@franz.ww.TU-Berlin.DE
    FULLNAME     Andreas König

cpan> a /and.*rt/
Author          ANDYD (Andy Dougherty)
Author          MERLYN (Randal L. Schwartz)
```

make, test, install, clean modules or distributions

These commands take any number of arguments and investigate what is necessary to perform the action. If the argument is a distribution file name (recognized by embedded slashes), it is processed. If it is a module, CPAN determines the distribution file in which this module is included and processes that.

Any `make` or `test` are run unconditionally. An

```
install <distribution_file>
```

also is run unconditionally. But for

```
install <module>
```

CPAN checks if an install is actually needed for it and prints *module up to date* in the case that the distribution file containing the module doesn't need to be updated.

CPAN also keeps track of what it has done within the current session and doesn't try to build a package a second time regardless if it succeeded or not. The `force` command takes as a first argument the method to invoke (currently: `make`, `test`, or `install`) and executes the command from scratch.

Example:

```
cpan> install OpenGL
OpenGL is up to date.
cpan> force install OpenGL
Running make
OpenGL-0.4/
OpenGL-0.4/COPYRIGHT
[...]
```

A `clean` command results in a

```
make clean
```

being executed within the distribution file's working directory.

readme, look module or distribution

These two commands take only one argument, be it a module or a distribution file. `readme` unconditionally runs, displaying the README of the associated distribution file. `Look` gets and untars (if not yet done) the distribution file, changes to the appropriate directory and opens a subshell process in that directory.

Signals

CPAN.pm installs signal handlers for SIGINT and SIGTERM. While you are in the cpan-shell it is intended that you can press `^C` anytime and return to the cpan-shell prompt. A SIGTERM will cause the cpan-shell to clean up and leave the shell loop. You can emulate the effect of a SIGTERM by sending two consecutive SIGINTs, which usually means by pressing `^C` twice.

CPAN.pm ignores a SIGPIPE. If the user sets inactivity_timeout, a SIGALRM is used during the run of the `perl Makefile.PL` subprocess.

### CPAN::Shell

The commands that are available in the shell interface are methods in the package CPAN::Shell. If you enter the shell command, all your input is split by the *Text::ParseWords::shellwords()* routine which acts like most shells do. The first word is being interpreted as the method to be called and the rest of the words are treated as arguments to this method. Continuation lines are supported if a line ends with a literal backslash.

### autobundle

`autobundle` writes a bundle file into the `$CPAN::Config->{cpan_home}/Bundle` directory. The file contains a list of all modules that are both available from CPAN and currently installed within `@INC`. The name of the bundle file is based on the current date and a counter.

### recompile

*recompile()* is a very special command in that it takes no argument and runs the make/test/install cycle with brute force over all installed dynamically loadable extensions (aka XS modules) with 'force' in effect. The primary purpose of this command is to finish a network installation. Imagine, you have a common source tree for two different architectures. You decide to do a completely independent fresh installation. You start on one architecture with the help of a Bundle file produced earlier. CPAN installs the whole Bundle for you, but when you try to repeat the job on the second architecture, CPAN responds with a `"Foo up to date"` message for all modules. So you invoke CPAN's recompile on the second architecture and you're done.

Another popular use for `recompile` is to act as a rescue in case your perl breaks binary compatibility. If one of the modules that CPAN uses is in turn depending on binary compatibility (so you cannot run CPAN commands), then you should try the CPAN::Nox module for recovery.

### The four `CPAN::*` Classes: Author, Bundle, Module, Distribution

Although it may be considered internal, the class hierarchy does matter for both users and programmer. CPAN.pm deals with above mentioned four classes, and all those classes share a set of methods. A classical single polymorphism is in effect. A metaclass object registers all objects of all kinds and indexes them with a string. The strings referencing objects have a separated namespace (well, not completely separated):

```
         Namespace                         Class

   words containing a "/" (slash)      Distribution
    words starting with Bundle::          Bundle
          everything else            Module or Author
```

Modules know their associated Distribution objects. They always refer to the most recent official release. Developers may mark their releases as unstable development versions (by inserting an underbar into the visible version number), so the really hottest and newest distribution file is not always the default. If a module Foo circulates on CPAN in both version 1.23 and 1.23_90, CPAN.pm offers a convenient way to install version 1.23 by saying

```
    install Foo
```

This would install the complete distribution file (say BAR/Foo-1.23.tar.gz) with all accompanying material. But if you would like to install version 1.23_90, you need to know where the distribution file resides on CPAN relative to the authors/id/ directory. If the author is BAR, this might be BAR/Foo-1.23_90.tar.gz; so you would have to say

```
    install BAR/Foo-1.23_90.tar.gz
```

The first example will be driven by an object of the class CPAN::Module, the second by an object of class CPAN::Distribution.

### Programmer's interface

If you do not enter the shell, the available shell commands are both available as methods (`CPAN::Shell->install(...)`) and as functions in the calling package (`install(...)`).

There's currently only one class that has a stable interface − CPAN::Shell. All commands that are available in the CPAN shell are methods of the class CPAN::Shell. Each of the commands that produce listings of modules (`r`, `autobundle`, `u`) returns a list of the IDs of all modules within the list.

expand($type,@things)
: The IDs of all objects available within a program are strings that can be expanded to the corresponding real objects with the `CPAN::Shell->expand("Module",@things)` method. Expand returns a list of CPAN::Module objects according to the `@things` arguments given. In scalar context it only returns the first element of the list.

Programming Examples
: This enables the programmer to do operations that combine functionalities that are available in the shell.

```
# install everything that is outdated on my disk:
perl -MCPAN -e 'CPAN::Shell->install(CPAN::Shell->r)'

# install my favorite programs if necessary:
for $mod (qw(Net::FTP MD5 Data::Dumper)){
    my $obj = CPAN::Shell->expand('Module',$mod);
    $obj->install;
}

# list all modules on my disk that have no VERSION number
for $mod (CPAN::Shell->expand("Module","/./")){
    next unless $mod->inst_file;
    # MakeMaker convention for undefined $VERSION:
    next unless $mod->inst_version eq "undef";
    print "No VERSION in ", $mod->id, "\n";
}
```

### Methods in the four

### Cache Manager

Currently the cache manager only keeps track of the build directory ($CPAN::Config->{build_dir}). It is a simple FIFO mechanism that deletes complete directories below `build_dir` as soon as the size of all directories there gets bigger than `$CPAN::Config`−>{build_cache} (in MB). The contents of this cache may be used for later re-installations that you intend to do manually, but will never be trusted by CPAN itself. This is due to the fact that the user might use these directories for building modules on different architectures.

There is another directory ($CPAN::Config->{keep_source_where}) where the original distribution files are kept. This directory is not covered by the cache manager and must be controlled by the user. If you choose to have the same directory as build_dir and as keep_source_where directory, then your sources will be deleted with the same fifo mechanism.

### Bundles

A bundle is just a perl module in the namespace Bundle:: that does not define any functions or methods. It usually only contains documentation.

It starts like a perl module with a package declaration and a `$VERSION` variable. After that the pod section looks like any other pod with the only difference being that *one special pod section* exists starting with (verbatim):

```
=head1 CONTENTS
```

In this pod section each line obeys the format

```
Module_Name [Version_String] [- optional text]
```

The only required part is the first field, the name of a module (e.g. Foo::Bar, ie. *not* the name of the distribution file). The rest of the line is optional. The comment part is delimited by a dash just as in the man page header.

The distribution of a bundle should follow the same convention as other distributions.

Bundles are treated specially in the CPAN package. If you say 'install Bundle::Tkkit' (assuming such a bundle exists), CPAN will install all the modules in the CONTENTS section of the pod. You can install your own Bundles locally by placing a conformant Bundle file somewhere into your `@INC` path. The *autobundle()* command which is available in the shell interface does that for you by including all currently installed modules in a snapshot bundle file.

### Prerequisites

If you have a local mirror of CPAN and can access all files with "file:" URLs, then you only need a perl better than perl5.003 to run this module. Otherwise Net::FTP is strongly recommended. LWP may be required for non-UNIX systems or if your nearest CPAN site is associated with an URL that is not `ftp:`.

If you have neither Net::FTP nor LWP, there is a fallback mechanism implemented for an external ftp command or for an external lynx command.

### Finding packages and VERSION

This module presumes that all packages on CPAN

- declare their `$VERSION` variable in an easy to parse manner. This prerequisite can hardly be relaxed because it consumes far too much memory to load all packages into the running program just to determine the `$VERSION` variable. Currently all programs that are dealing with version use something like this

  ```
  perl -MExtUtils::MakeMaker -le \
      'print MM->parse_version($ARGV[0])' filename
  ```

  If you are author of a package and wonder if your `$VERSION` can be parsed, please try the above method.
- come as compressed or gzipped tarfiles or as zip files and contain a Makefile.PL (well, we try to handle a bit more, but without much enthusiasm).

### Debugging

The debugging of this module is pretty difficult, because we have interferences of the software producing the indices on CPAN, of the mirroring process on CPAN, of packaging, of configuration, of synchronicity, and of bugs within CPAN.pm.

In interactive mode you can try "o debug" which will list options for debugging the various parts of the package. The output may not be very useful for you as it's just a by-product of my own testing, but if you have an idea which part of the package may have a bug, it's sometimes worth to give it a try and send me more specific output. You should know that "o debug" has built-in completion support.

### Floppy, Zip, and all that Jazz

CPAN.pm works nicely without network too. If you maintain machines that are not networked at all, you should consider working with file: URLs. Of course, you have to collect your modules somewhere first. So you might use CPAN.pm to put together all you need on a networked machine. Then copy the `$CPAN::Config->`{keep_source_where} (but not `$CPAN::Config->`{build_dir}) directory on a floppy. This floppy is kind of a personal CPAN. CPAN.pm on the non-networked machines works nicely with this floppy.

**CONFIGURATION**

When the CPAN module is installed, a site wide configuration file is created as CPAN/Config.pm. The default values defined there can be overridden in another configuration file: CPAN/MyConfig.pm. You can store this file in `$HOME`/.cpan/CPAN/MyConfig.pm if you want, because `$HOME`/.cpan is added to the search path of the CPAN module before the *use()* or *require()* statements.

Currently the following keys in the hash reference `$CPAN::Config` are defined:

```
build_cache        size of cache for directories to build modules
build_dir          locally accessible directory to build modules
index_expire       after this many days refetch index files
cpan_home          local directory reserved for this package
gzip               location of external program gzip
inactivity_timeout breaks interactive Makefile.PLs after this
                   many seconds inactivity. Set to 0 to never break.
inhibit_startup_message
                   if true, does not print the startup message
keep_source        keep the source in a local directory?
keep_source_where  directory in which to keep the source (if we do)
make               location of external make program
make_arg           arguments that should always be passed to 'make'
make_install_arg   same as make_arg for 'make install'
makepl_arg         arguments passed to 'perl Makefile.PL'
pager              location of external program more (or any pager)
tar                location of external program tar
unzip              location of external program unzip
urllist            arrayref to nearby CPAN sites (or equivalent locations)
wait_list          arrayref to a wait server to try (See CPAN::WAIT)
```

You can set and query each of these options interactively in the cpan shell with the command set defined within the `o conf` command:

o conf <scalar option>
: prints the current value of the *scalar option*

o conf <scalar option> <value>
: Sets the value of the *scalar option* to *value*

o conf <list option>
: prints the current value of the *list option* in MakeMaker's neatvalue format.

o conf <list option> [shift|pop]
: shifts or pops the array in the *list option* variable

o conf <list option> [unshift|push|splice] <list>
: works like the corresponding perl commands.

**CD−ROM support**

The `urllist` parameter of the configuration table contains a list of URLs that are to be used for downloading. If the list contains any `file` URLs, CPAN always tries to get files from there first. This feature is disabled for index files. So the recommendation for the owner of a CD−ROM with CPAN contents is: include your local, possibly outdated CD−ROM as a `file` URL at the end of urllist, e.g.

```
o conf urllist push file://localhost/CDROM/CPAN
```

CPAN.pm will then fetch the index files from one of the CPAN sites that come at the beginning of urllist. It will later check for each module if there is a local copy of the most recent version.

**SECURITY**

There's no strong security layer in CPAN.pm. CPAN.pm helps you to install foreign, unmasked, unsigned code on your machine. We compare to a checksum that comes from the net just as the distribution file itself. If somebody has managed to tamper with the distribution file, they may have as well tampered with the CHECKSUMS file. Future development will go towards strong authentification.

**EXPORT**

Most functions in package CPAN are exported per default. The reason for this is that the primary use is intended for the cpan shell or for oneliners.

**BUGS**

We should give coverage for _all_ of the CPAN and not just the PAUSE part, right? In this discussion CPAN and PAUSE have become equal -- but they are not. PAUSE is authors/ and modules/. CPAN is PAUSE plus the clpa/, doc/, misc/, ports/, src/, scripts/.

Future development should be directed towards a better integration of the other parts.

If a Makefile.PL requires special customization of libraries, prompts the user for special input, etc. then you may find CPAN is not able to build the distribution. In that case, you should attempt the traditional method of building a Perl module package from a shell.

**AUTHOR**

Andreas König <a.koenig@mind.de>

**SEE ALSO**

*perl* (1), *CPAN::Nox* (3)

## NAME

CPAN::FirstTime − Utility for CPAN::Config file Initialization

## SYNOPSIS

*CPAN::FirstTime::init()*

## DESCRIPTION

The init routine asks a few questions and writes a CPAN::Config file. Nothing special.

## NAME

CPAN::Nox – Wrapper around CPAN.pm without using any XS module

## SYNOPSIS

Interactive mode:

```
perl -MCPAN::Nox -e shell;
```

## DESCRIPTION

This package has the same functionality as CPAN.pm, but tries to prevent the usage of compiled extensions during it's own execution. It's primary purpose is a rescue in case you upgraded perl and broke binary compatibility somehow.

## SEE ALSO

*CPAN*(3)

## NAME

getcwd − get pathname of current working directory

## SYNOPSIS

```
use Cwd;
$dir = cwd;

use Cwd;
$dir = getcwd;

use Cwd;
$dir = fastgetcwd;

use Cwd 'chdir';
chdir "/tmp";
print $ENV{'PWD'};

use Cwd 'abs_path';
print abs_path($ENV{'PWD'});

use Cwd 'fast_abs_path';
print fast_abs_path($ENV{'PWD'});
```

## DESCRIPTION

The *getcwd()* function re-implements the *getcwd*(3) (or *getwd*(3)) functions in Perl.

The *abs_path()* function takes a single argument and returns the absolute pathname for that argument. It uses the same algoritm as *getcwd()*. (actually *getcwd()* is *abs_path*(''."))

The *fastcwd()* function looks the same as *getcwd()*, but runs faster. It's also more dangerous because it might conceivably *chdir()* you out of a directory that it can't *chdir()* you back into. If fastcwd encounters a problem it will return undef but will probably leave you in a different directory. For a measure of extra security, if everything appears to have worked, the *fastcwd()* function will check that it leaves you in the same directory that it started in. If it has changed it will `die` with the message ''Unstable directory path, current directory changed unexpectedly''. That should never happen.

The *fast_abs_path()* function looks the same as *abs_path()*, but runs faster. And like *fastcwd()* is more dangerous.

The *cwd()* function looks the same as getcwd and fastgetcwd but is implemented using the most natural and safe form for the current architecture. For most systems it is identical to `pwd` (but without the trailing line terminator).

It is recommended that cwd (or another **cwd()* function) is used in *all* code to ensure portability.

If you ask to override your *chdir()* built-in function, then your PWD environment variable will be kept up to date. (See the section on *Overriding Builtin Functions* in the *perlsub* manpage.) Note that it will only be kept up to date if all packages which use chdir import it from Cwd.

**NAME**

Data::Dumper – stringified perl data structures, suitable for both printing and `eval`

**SYNOPSIS**

```
use Data::Dumper;

# simple procedural interface
print Dumper($foo, $bar);

# extended usage with names
print Data::Dumper->Dump([$foo, $bar], [qw(foo *ary)]);

# configuration variables
{
  local $Data::Dump::Purity = 1;
  eval Data::Dumper->Dump([$foo, $bar], [qw(foo *ary)]);
}

# OO usage
$d = Data::Dumper->new([$foo, $bar], [qw(foo *ary)]);
   ...
print $d->Dump;
   ...
$d->Purity(1)->Terse(1)->Deepcopy(1);
eval $d->Dump;
```

**DESCRIPTION**

Given a list of scalars or reference variables, writes out their contents in perl syntax. The references can also be objects. The contents of each variable is output in a single Perl statement. Handles self-referential structures correctly.

The return value can be `eval`ed to get back an identical copy of the original reference structure.

Any references that are the same as one of those passed in will be named `$VAR`*n* (where *n* is a numeric suffix), and other duplicate references to substructures within `$VAR`*n* will be appropriately labeled using arrow notation. You can specify names for individual values to be dumped if you use the `Dump()` method, or you can change the default `$VAR` prefix to something else. See `$Data::Dumper::Varname` and `$Data::Dumper::Terse` below.

The default output of self-referential structures can be `eval`ed, but the nested references to `$VAR`*n* will be undefined, since a recursive structure cannot be constructed using one Perl statement. You should set the `Purity` flag to 1 to get additional statements that will correctly fill in these references.

In the extended usage form, the references to be dumped can be given user-specified names. If a name begins with a `*`, the output will describe the dereferenced type of the supplied reference for hashes and arrays, and coderefs. Output of names will be avoided where possible if the `Terse` flag is set.

In many cases, methods that are used to set the internal state of the object will return the object itself, so method calls can be conveniently chained together.

Several styles of output are possible, all controlled by setting the `Indent` flag. See the section on *Configuration Variables or Methods* below for details.

### Methods

*PACKAGE*–>new(*ARRAYREF [*, *ARRAYREF]*)

Returns a newly created `Data::Dumper` object. The first argument is an anonymous array of values to be dumped. The optional second argument is an anonymous array of names for the values. The names need not have a leading `$` sign, and must be comprised of alphanumeric characters. You can begin a name with a `*` to specify that the dereferenced type must be dumped instead of the reference itself, for ARRAY and HASH references.

The prefix specified by `$Data::Dumper::Varname` will be used with a numeric suffix if the name for a value is undefined.

Data::Dumper will catalog all references encountered while dumping the values. Cross-references (in the form of names of substructures in perl syntax) will be inserted at all possible points, preserving any structural interdependencies in the original set of values. Structure traversal is depth-first, and proceeds in order from the first supplied value to the last.

*$OBJ*−>Dump *or PACKAGE*−>Dump(*ARRAYREF [*, *ARRAYREF]*)
: Returns the stringified form of the values stored in the object (preserving the order in which they were supplied to `new`), subject to the configuration options below. In an array context, it returns a list of strings corresponding to the supplied values.

  The second form, for convenience, simply calls the `new` method on its arguments before dumping the object immediately.

*$OBJ*−>Dumpxs *or PACKAGE*−>Dumpxs(*ARRAYREF [*, *ARRAYREF]*)
: This method is available if you were able to compile and install the XSUB extension to `Data::Dumper`. It is exactly identical to the `Dump` method above, only about 4 to 5 times faster, since it is written entirely in C.

*$OBJ*−>Seen(*[HASHREF]*)
: Queries or adds to the internal table of already encountered references. You must use `Reset` to explicitly clear the table if needed. Such references are not dumped; instead, their names are inserted wherever they are encountered subsequently. This is useful especially for properly dumping subroutine references.

  Expects a anonymous hash of name => value pairs. Same rules apply for names as in `new`. If no argument is supplied, will return the "seen" list of name => value pairs, in an array context. Otherwise, returns the object itself.

*$OBJ*−>Values(*[ARRAYREF]*)
: Queries or replaces the internal array of values that will be dumped. When called without arguments, returns the values. Otherwise, returns the object itself.

*$OBJ*−>Names(*[ARRAYREF]*)
: Queries or replaces the internal array of user supplied names for the values that will be dumped. When called without arguments, returns the names. Otherwise, returns the object itself.

*$OBJ*−>Reset
: Clears the internal table of "seen" references and returns the object itself.

## Functions

Dumper(*LIST*)
: Returns the stringified form of the values in the list, subject to the configuration options below. The values will be named `$VAR`*n* in the output, where *n* is a numeric suffix. Will return a list of strings in an array context.

DumperX(*LIST*)
: Identical to the `Dumper()` function above, but this calls the XSUB implementation. Only available if you were able to compile and install the XSUB extensions in `Data::Dumper`.

## Configuration Variables or Methods

Several configuration variables can be used to control the kind of output generated when using the procedural interface. These variables are usually `local`ized in a block so that other parts of the code are not affected by the change.

These variables determine the default state of the object created by calling the `new` method, but cannot be used to alter the state of the object thereafter. The equivalent method names should be used instead to query or set the internal state of the object.

The method forms return the object itself when called with arguments, so that they can be chained together nicely.

$Data::Dumper::Indent *or $OBJ*–>Indent(*[NEWVAL]*)
: Controls the style of indentation. It can be set to 0, 1, 2 or 3. Style 0 spews output without any newlines, indentation, or spaces between list items. It is the most compact format possible that can still be called valid perl. Style 1 outputs a readable form with newlines but no fancy indentation (each level in the structure is simply indented by a fixed amount of whitespace). Style 2 (the default) outputs a very readable form which takes into account the length of hash keys (so the hash value lines up). Style 3 is like style 2, but also annotates the elements of arrays with their index (but the comment is on its own line, so array output consumes twice the number of lines). Style 2 is the default.

$Data::Dumper::Purity *or $OBJ*–>Purity(*[NEWVAL]*)
: Controls the degree to which the output can be `eval`ed to recreate the supplied reference structures. Setting it to 1 will output additional perl statements that will correctly recreate nested references. The default is 0.

$Data::Dumper::Pad *or $OBJ*–>Pad(*[NEWVAL]*)
: Specifies the string that will be prefixed to every line of the output. Empty string by default.

$Data::Dumper::Varname *or $OBJ*–>Varname(*[NEWVAL]*)
: Contains the prefix to use for tagging variable names in the output. The default is "VAR".

$Data::Dumper::Useqq *or $OBJ*–>Useqq(*[NEWVAL]*)
: When set, enables the use of double quotes for representing string values. Whitespace other than space will be represented as `[\n\t\r]`, "unsafe" characters will be backslashed, and unprintable characters will be output as quoted octal integers. Since setting this variable imposes a performance penalty, the default is 0. The `Dumpxs()` method does not honor this flag yet.

$Data::Dumper::Terse *or $OBJ*–>Terse(*[NEWVAL]*)
: When set, Data::Dumper will emit single, non-self-referential values as atoms/terms rather than statements. This means that the `$VAR`*n* names will be avoided where possible, but be advised that such output may not always be parseable by `eval`.

$Data::Dumper::Freezer *or* $*OBJ*–>Freezer(*[NEWVAL]*)
: Can be set to a method name, or to an empty string to disable the feature. Data::Dumper will invoke that method via the object before attempting to stringify it. This method can alter the contents of the object (if, for instance, it contains data allocated from C), and even rebless it in a different package. The client is responsible for making sure the specified method can be called via the object, and that the object ends up containing only perl data types after the method has been called. Defaults to an empty string.

$Data::Dumper::Toaster *or* $*OBJ*–>Toaster(*[NEWVAL]*)
: Can be set to a method name, or to an empty string to disable the feature. Data::Dumper will emit a method call for any objects that are to be dumped using the syntax `bless(DATA, CLASS)`–*METHOD()*>. Note that this means that the method specified will have to perform any modifications required on the object (like creating new state within it, and/or reblessing it in a different package) and then return it. The client is responsible for making sure the method can be called via the object, and that it returns a valid object. Defaults to an empty string.

$Data::Dumper::Deepcopy *or* $*OBJ*–>Deepcopy(*[NEWVAL]*)
: Can be set to a boolean value to enable deep copies of structures. Cross-referencing will then only be done when absolutely essential (i.e., to break reference cycles). Default is 0.

$Data::Dumper::Quotekeys *or* $*OBJ*–>Quotekeys(*[NEWVAL]*)
: Can be set to a boolean value to control whether hash keys are quoted. A false value will avoid quoting hash keys when it looks like a simple string. Default is 1, which will always enclose hash keys in

quotes.

$Data::Dumper::Bless *or* $*OBJ*–>Bless(*[NEWVAL]*)
: Can be set to a string that specifies an alternative to the `bless` builtin operator used to create objects. A function with the specified name should exist, and should accept the same arguments as the builtin. Default is `bless`.

### Exports

Dumper

## EXAMPLES

Run these code snippets to get a quick feel for the behavior of this module. When you are through with these examples, you may want to add or change the various configuration variables described above, to see their behavior. (See the testsuite in the Data::Dumper distribution for more examples.)

```
use Data::Dumper;

package Foo;
sub new {bless {'a' => 1, 'b' => sub { return "foo" }}, $_[0]};

package Fuz;                       # a weird REF-REF-SCALAR object
sub new {bless \($_ = \ 'fu\'z'), $_[0]};

package main;
$foo = Foo->new;
$fuz = Fuz->new;
$boo = [ 1, [], "abcd", \*foo,
         {1 => 'a', 023 => 'b', 0x45 => 'c'},
         \\"p\q\'r", $foo, $fuz];

########
# simple usage
########

$bar = eval(Dumper($boo));
print($@) if $@;
print Dumper($boo), Dumper($bar);  # pretty print (no array indices)

$Data::Dumper::Terse = 1;          # don't output names where feasible
$Data::Dumper::Indent = 0;         # turn off all pretty print
print Dumper($boo), "\n";

$Data::Dumper::Indent = 1;         # mild pretty print
print Dumper($boo);

$Data::Dumper::Indent = 3;         # pretty print with array indices
print Dumper($boo);
```

```
$Data::Dumper::Useqq = 1;          # print strings in double quotes
print Dumper($boo);


########
# recursive structures
########

@c = ('c');
$c = \@c;
$b = {};
$a = [1, $b, $c];
$b->{a} = $a;
$b->{b} = $a->[1];
$b->{c} = $a->[2];
print Data::Dumper->Dump([$a,$b,$c], [qw(a b c)]);


$Data::Dumper::Purity = 1;         # fill in the holes for eval
print Data::Dumper->Dump([$a, $b], [qw(*a b)]); # print as @a
print Data::Dumper->Dump([$b, $a], [qw(*b a)]); # print as %b


$Data::Dumper::Deepcopy = 1;       # avoid cross-refs
print Data::Dumper->Dump([$b, $a], [qw(*b a)]);


$Data::Dumper::Purity = 0;         # avoid cross-refs
print Data::Dumper->Dump([$b, $a], [qw(*b a)]);


########
# object-oriented usage
########

$d = Data::Dumper->new([$a,$b], [qw(a b)]);
$d->Seen({'*c' => $c});            # stash a ref without printing it
$d->Indent(3);
print $d->Dump;
$d->Reset->Purity(0);              # empty the seen cache
print join "----\n", $d->Dump;


########
# persistence
########

package Foo;
sub new { bless { state => 'awake' }, shift }
sub Freeze {
    my $s = shift;
    print STDERR "preparing to sleep\n";
    $s->{state} = 'asleep';
    return bless $s, 'Foo::ZZZ';
```

```
}

package Foo::ZZZ;
sub Thaw {
    my $s = shift;
    print STDERR "waking up\n";
    $s->{state} = 'awake';
    return bless $s, 'Foo';
}

package Foo;
use Data::Dumper;
$a = Foo->new;
$b = Data::Dumper->new([$a], ['c']);
$b->Freezer('Freeze');
$b->Toaster('Thaw');
$c = $b->Dump;
print $c;
$d = eval $c;
print Data::Dumper->Dump([$d], ['d']);


########
# symbol substitution (useful for recreating CODE refs)
########

sub foo { print "foo speaking\n" }
*other = \&foo;
$bar = [ \&other ];
$d = Data::Dumper->new([\&other,$bar],['*other','bar']);
$d->Seen({ '*foo' => \&foo });
print $d->Dump;
```

**BUGS**

Due to limitations of Perl subroutine call semantics, you cannot pass an array or hash. Prepend it with a `\` to pass its reference instead. This will be remedied in time, with the arrival of prototypes in later versions of Perl. For now, you need to use the extended usage form, and prepend the name with a `*` to output it as a hash or array.

`Data::Dumper` cheats with CODE references. If a code reference is encountered in the structure being processed, an anonymous subroutine that contains the string '"DUMMY"' will be inserted in its place, and a warning will be printed if `Purity` is set. You can `eval` the result, but bear in mind that the anonymous sub that gets created is just a placeholder. Someday, perl will have a switch to cache-on-demand the string representation of a compiled piece of code, I hope. If you have prior knowledge of all the code refs that your data structures are likely to have, you can use the `Seen` method to pre-seed the internal reference table and make the dumped output point to them, instead. See the *EXAMPLES* manpage above.

The `Useqq` flag is not honored by `Dumpxs()` (it always outputs strings in single quotes).

SCALAR objects have the weirdest looking `bless` workaround.

**AUTHOR**

Gurusamy Sarathy gsar@umich.edu

**VERSION**

Version 2.09 (9 July 1998)

**SEE ALSO**

*perl*(1)

## NAME

DB_File − Perl5 access to Berkeley DB version 1.x

## SYNOPSIS

```
use DB_File ;

[$X =] tie %hash,  'DB_File', [$filename, $flags, $mode, $DB_HASH] ;
[$X =] tie %hash,  'DB_File', $filename, $flags, $mode, $DB_BTREE ;
[$X =] tie @array, 'DB_File', $filename, $flags, $mode, $DB_RECNO ;

$status = $X->del($key [, $flags]) ;
$status = $X->put($key, $value [, $flags]) ;
$status = $X->get($key, $value [, $flags]) ;
$status = $X->seq($key, $value, $flags) ;
$status = $X->sync([$flags]) ;
$status = $X->fd ;

# BTREE only
$count = $X->get_dup($key) ;
@list  = $X->get_dup($key) ;
%list  = $X->get_dup($key, 1) ;

# RECNO only
$a = $X->length;
$a = $X->pop ;
$X->push(list);
$a = $X->shift;
$X->unshift(list);

untie %hash ;
untie @array ;
```

## DESCRIPTION

**DB_File** is a module which allows Perl programs to make use of the facilities provided by Berkeley DB version 1.x (if you have a newer version of DB, see the section on *Using DB_File with Berkeley DB version 2*). It is assumed that you have a copy of the Berkeley DB manual pages at hand when reading this documentation. The interface defined here mirrors the Berkeley DB interface closely.

Berkeley DB is a C library which provides a consistent interface to a number of database formats. **DB_File** provides an interface to all three of the database types currently supported by Berkeley DB.

The file types are:

**DB_HASH**

This database type allows arbitrary key/value pairs to be stored in data files. This is equivalent to the functionality provided by other hashing packages like DBM, NDBM, ODBM, GDBM, and SDBM. Remember though, the files created using DB_HASH are not compatible with any of the other packages mentioned.

A default hashing algorithm, which will be adequate for most applications, is built into Berkeley DB. If you do need to use your own hashing algorithm it is possible to write your own in Perl and have **DB_File** use it instead.

**DB_BTREE**

The btree format allows arbitrary key/value pairs to be stored in a sorted, balanced binary tree.

As with the DB_HASH format, it is possible to provide a user defined Perl routine to perform the comparison of keys. By default, though, the keys are stored in lexical order.

**DB_RECNO**

DB_RECNO allows both fixed-length and variable-length flat text files to be manipulated using the same key/value pair interface as in DB_HASH and DB_BTREE. In this case the key will consist of a record (line) number.

### Using DB_File with Berkeley DB version 2

Although **DB_File** is intended to be used with Berkeley DB version 1, it can also be used with version 2. In this case the interface is limited to the functionality provided by Berkeley DB 1.x. Anywhere the version 2 interface differs, **DB_File** arranges for it to work like version 1. This feature allows **DB_File** scripts that were built with version 1 to be migrated to version 2 without any changes.

If you want to make use of the new features available in Berkeley DB 2.x, use the Perl module **BerkeleyDB** instead.

At the time of writing this document the **BerkeleyDB** module is still alpha quality (the version number is < 1.0), and so unsuitable for use in any serious development work. Once its version number is >= 1.0, it is considered stable enough for real work.

**Note:** The database file format has changed in Berkeley DB version 2. If you cannot recreate your databases, you must dump any existing databases with the `db_dump185` utility that comes with Berkeley DB. Once you have upgraded DB_File to use Berkeley DB version 2, your databases can be recreated using `db_load`. Refer to the Berkeley DB documentation for further details.

Please read the *COPYRIGHT* manpage before using version 2.x of Berkeley DB with DB_File.

### Interface to Berkeley DB

**DB_File** allows access to Berkeley DB files using the *tie()* mechanism in Perl 5 (for full details, see the `tie()` entry in the *perlfunc* manpage). This facility allows **DB_File** to access Berkeley DB files using either an associative array (for DB_HASH & DB_BTREE file types) or an ordinary array (for the DB_RECNO file type).

In addition to the *tie()* interface, it is also possible to access most of the functions provided in the Berkeley DB API directly. See the section on *THE API INTERFACE*.

### Opening a Berkeley DB Database File

Berkeley DB uses the function *dbopen()* to open or create a database. Here is the C prototype for *dbopen()*:

```
DB*
dbopen (const char * file, int flags, int mode,
        DBTYPE type, const void * openinfo)
```

The parameter `type` is an enumeration which specifies which of the 3 interface methods (DB_HASH, DB_BTREE or DB_RECNO) is to be used. Depending on which of these is actually chosen, the final parameter, *openinfo* points to a data structure which allows tailoring of the specific interface method.

This interface is handled slightly differently in **DB_File**. Here is an equivalent call using **DB_File**:

```
tie %array, 'DB_File', $filename, $flags, $mode, $DB_HASH ;
```

The `filename`, `flags` and `mode` parameters are the direct equivalent of their *dbopen()* counterparts. The final parameter `$DB_HASH` performs the function of both the `type` and `openinfo` parameters in *dbopen()*.

In the example above `$DB_HASH` is actually a pre-defined reference to a hash object. **DB_File** has three of these pre-defined references. Apart from `$DB_HASH`, there is also `$DB_BTREE` and `$DB_RECNO`.

The keys allowed in each of these pre-defined references is limited to the names used in the equivalent C structure. So, for example, the `$DB_HASH` reference will only allow keys called `bsize`, `cachesize`, `ffactor`, `hash`, `lorder` and `nelem`.

To change one of these elements, just assign to it like this:

```
$DB_HASH->{'cachesize'} = 10000 ;
```

The three predefined variables `$DB_HASH`, `$DB_BTREE` and `$DB_RECNO` are usually adequate for most applications. If you do need to create extra instances of these objects, constructors are available for each file type.

Here are examples of the constructors and the valid options available for DB_HASH, DB_BTREE and DB_RECNO respectively.

```
$a = new DB_File::HASHINFO ;
$a->{'bsize'} ;
$a->{'cachesize'} ;
$a->{'ffactor'};
$a->{'hash'} ;
$a->{'lorder'} ;
$a->{'nelem'} ;

$b = new DB_File::BTREEINFO ;
$b->{'flags'} ;
$b->{'cachesize'} ;
$b->{'maxkeypage'} ;
$b->{'minkeypage'} ;
$b->{'psize'} ;
$b->{'compare'} ;
$b->{'prefix'} ;
$b->{'lorder'} ;

$c = new DB_File::RECNOINFO ;
$c->{'bval'} ;
$c->{'cachesize'} ;
$c->{'psize'} ;
$c->{'flags'} ;
$c->{'lorder'} ;
$c->{'reclen'} ;
$c->{'bfname'} ;
```

The values stored in the hashes above are mostly the direct equivalent of their C counterpart. Like their C counterparts, all are set to a default values – that means you don't have to set *all* of the values when you only want to change one. Here is an example:

```
$a = new DB_File::HASHINFO ;
$a->{'cachesize'} =  12345 ;
tie %y, 'DB_File', "filename", $flags, 0777, $a ;
```

A few of the options need extra discussion here. When used, the C equivalent of the keys `hash`, `compare` and `prefix` store pointers to C functions. In **DB_File** these keys are used to store references to Perl subs. Below are templates for each of the subs:

```
sub hash
{
    my ($data) = @_ ;
    ...
    # return the hash value for $data
    return $hash ;
}
```

```
sub compare
{
    my ($key, $key2) = @_ ;
    ...
    # return  0 if $key1 eq $key2
    #        -1 if $key1 lt $key2
    #         1 if $key1 gt $key2
    return (-1 , 0 or 1) ;
}

sub prefix
{
    my ($key, $key2) = @_ ;
    ...
    # return number of bytes of $key2 which are
    # necessary to determine that it is greater than $key1
    return $bytes ;
}
```

See the section on *Changing the BTREE sort order* for an example of using the `compare` template.

If you are using the DB_RECNO interface and you intend making use of `bval`, you should check out the section on *The 'bval' Option*.

### Default Parameters

It is possible to omit some or all of the final 4 parameters in the call to `tie` and let them take default values. As DB_HASH is the most common file format used, the call:

```
tie %A, "DB_File", "filename" ;
```

is equivalent to:

```
tie %A, "DB_File", "filename", O_CREAT|O_RDWR, 0666, $DB_HASH ;
```

It is also possible to omit the filename parameter as well, so the call:

```
tie %A, "DB_File" ;
```

is equivalent to:

```
tie %A, "DB_File", undef, O_CREAT|O_RDWR, 0666, $DB_HASH ;
```

See the section on *In Memory Databases* for a discussion on the use of `undef` in place of a filename.

### In Memory Databases

Berkeley DB allows the creation of in-memory databases by using NULL (that is, a `(char *)0` in C) in place of the filename. **DB_File** uses `undef` instead of NULL to provide this functionality.

## DB_HASH

The DB_HASH file format is probably the most commonly used of the three file formats that **DB_File** supports. It is also very straightforward to use.

### A Simple Example

This example shows how to create a database, add key/value pairs to the database, delete keys/value pairs and finally how to enumerate the contents of the database.

```
use strict ;
use DB_File ;
use vars qw( %h $k $v ) ;

tie %h, "DB_File", "fruit", O_RDWR|O_CREAT, 0640, $DB_HASH
    or die "Cannot open file 'fruit': $!\n";

# Add a few key/value pairs to the file
$h{"apple"} = "red" ;
$h{"orange"} = "orange" ;
$h{"banana"} = "yellow" ;
$h{"tomato"} = "red" ;

# Check for existence of a key
print "Banana Exists\n\n" if $h{"banana"} ;

# Delete a key/value pair.
delete $h{"apple"} ;

# print the contents of the file
while (($k, $v) = each %h)
  { print "$k -> $v\n" }

untie %h ;
```

here is the output:

```
Banana Exists

orange -> orange
tomato -> red
banana -> yellow
```

Note that the like ordinary associative arrays, the order of the keys retrieved is in an apparently random order.

**DB_BTREE**

The DB_BTREE format is useful when you want to store data in a given order. By default the keys will be stored in lexical order, but as you will see from the example shown in the next section, it is very easy to define your own sorting function.

### Changing the BTREE sort order

This script shows how to override the default sorting algorithm that BTREE uses. Instead of using the normal lexical ordering, a case insensitive compare function will be used.

```
use strict ;
use DB_File ;

my %h ;

sub Compare
{
    my ($key1, $key2) = @_ ;
    "\L$key1" cmp "\L$key2" ;
}
```

```
# specify the Perl sub that will do the comparison
$DB_BTREE->{'compare'} = \&Compare ;

tie %h, "DB_File", "tree", O_RDWR|O_CREAT, 0640, $DB_BTREE
    or die "Cannot open file 'tree': $!\n" ;

# Add a key/value pair to the file
$h{'Wall'} = 'Larry' ;
$h{'Smith'} = 'John' ;
$h{'mouse'} = 'mickey' ;
$h{'duck'}  = 'donald' ;

# Delete
delete $h{"duck"} ;

# Cycle through the keys printing them in order.
# Note it is not necessary to sort the keys as
# the btree will have kept them in order automatically.
foreach (keys %h)
  { print "$_\n" }

untie %h ;
```

Here is the output from the code above.

```
mouse
Smith
Wall
```

There are a few point to bear in mind if you want to change the ordering in a BTREE database:

1. The new compare function must be specified when you create the database.
2. You cannot change the ordering once the database has been created. Thus you must use the same compare function every time you access the database.

**Handling Duplicate Keys**

The BTREE file type optionally allows a single key to be associated with an arbitrary number of values. This option is enabled by setting the flags element of `$DB_BTREE` to R_DUP when creating the database.

There are some difficulties in using the tied hash interface if you want to manipulate a BTREE database with duplicate keys. Consider this code:

```
use strict ;
use DB_File ;

use vars qw($filename %h ) ;
```

```
$filename = "tree" ;
unlink $filename ;

# Enable duplicate records
$DB_BTREE->{'flags'} = R_DUP ;

tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0640, $DB_BTREE
    or die "Cannot open $filename: $!\n";

# Add some key/value pairs to the file
$h{'Wall'} = 'Larry' ;
$h{'Wall'} = 'Brick' ; # Note the duplicate key
$h{'Wall'} = 'Brick' ; # Note the duplicate key and value
$h{'Smith'} = 'John' ;
$h{'mouse'} = 'mickey' ;

# iterate through the associative array
# and print each key/value pair.
foreach (keys %h)
  { print "$_  -> $h{$_}\n" }

untie %h ;
```

Here is the output:

```
Smith   -> John
Wall    -> Larry
Wall    -> Larry
Wall    -> Larry
mouse   -> mickey
```

As you can see 3 records have been successfully created with key `Wall` – the only thing is, when they are retrieved from the database they *seem* to have the same value, namely `Larry`. The problem is caused by the way that the associative array interface works. Basically, when the associative array interface is used to fetch the value associated with a given key, it will only ever retrieve the first value.

Although it may not be immediately obvious from the code above, the associative array interface can be used to write values with duplicate keys, but it cannot be used to read them back from the database.

The way to get around this problem is to use the Berkeley DB API method called `seq`. This method allows sequential access to key/value pairs. See the section on *THE API INTERFACE* for details of both the `seq` method and the API in general.

Here is the script above rewritten using the `seq` API method.

```
use strict ;
use DB_File ;

use vars qw($filename $x %h $status $key $value) ;
```

```
$filename = "tree" ;
unlink $filename ;

# Enable duplicate records
$DB_BTREE->{'flags'} = R_DUP ;

$x = tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0640, $DB_BTREE
    or die "Cannot open $filename: $!\n";

# Add some key/value pairs to the file
$h{'Wall'} = 'Larry' ;
$h{'Wall'} = 'Brick' ; # Note the duplicate key
$h{'Wall'} = 'Brick' ; # Note the duplicate key and value
$h{'Smith'} = 'John' ;
$h{'mouse'} = 'mickey' ;

# iterate through the btree using seq
# and print each key/value pair.
$key = $value = 0 ;
for ($status = $x->seq($key, $value, R_FIRST) ;
     $status == 0 ;
     $status = $x->seq($key, $value, R_NEXT) )
  {  print "$key -> $value\n" }

undef $x ;
untie %h ;
```

that prints:

```
Smith   -> John
Wall    -> Brick
Wall    -> Brick
Wall    -> Larry
mouse   -> mickey
```

This time we have got all the key/value pairs, including the multiple values associated with the key `Wall`.

**The** *get_dup()* Method

**DB_File** comes with a utility method, called `get_dup`, to assist in reading duplicate values from BTREE databases. The method can take the following forms:

```
$count = $x->get_dup($key) ;
@list  = $x->get_dup($key) ;
%list  = $x->get_dup($key, 1) ;
```

In a scalar context the method returns the number of values associated with the key, `$key`.

In list context, it returns all the values which match `$key`. Note that the values will be returned in an apparently random order.

In list context, if the second parameter is present and evaluates TRUE, the method returns an associative array. The keys of the associative array correspond to the values that matched in the BTREE and the values of the array are a count of the number of times that particular value occurred in the BTREE.

So assuming the database created above, we can use `get_dup` like this:

```
my $cnt  = $x->get_dup("Wall") ;
print "Wall occurred $cnt times\n" ;

my %hash = $x->get_dup("Wall", 1) ;
print "Larry is there\n" if $hash{'Larry'} ;
print "There are $hash{'Brick'} Brick Walls\n" ;

my @list = $x->get_dup("Wall") ;
print "Wall =>      [@list]\n" ;

@list = $x->get_dup("Smith") ;
print "Smith =>     [@list]\n" ;

@list = $x->get_dup("Dog") ;
print "Dog =>       [@list]\n" ;
```

and it will print:

```
Wall occurred 3 times
Larry is there
There are 2 Brick Walls
Wall =>     [Brick Brick Larry]
Smith =>    [John]
Dog =>      []
```

### Matching Partial Keys

The BTREE interface has a feature which allows partial keys to be matched. This functionality is *only* available when the `seq` method is used along with the R_CURSOR flag.

```
$x->seq($key, $value, R_CURSOR) ;
```

Here is the relevant quote from the dbopen man page where it defines the use of the R_CURSOR flag with seq:

```
Note, for the DB_BTREE access method, the returned key is not
necessarily an exact match for the specified key. The returned key
is the smallest key greater than or equal to the specified key,
permitting partial key matches and range searches.
```

In the example script below, the `match` sub uses this feature to find and print the first matching key/value pair given a partial key.

```
use strict ;
use DB_File ;
use Fcntl ;

use vars qw($filename $x %h $st $key $value) ;
```

```
sub match
{
    my $key = shift ;
    my $value = 0;
    my $orig_key = $key ;
    $x->seq($key, $value, R_CURSOR) ;
    print "$orig_key\t-> $key\t-> $value\n" ;
}

$filename = "tree" ;
unlink $filename ;

$x = tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0640, $DB_BTREE
    or die "Cannot open $filename: $!\n";

# Add some key/value pairs to the file
$h{'mouse'} = 'mickey' ;
$h{'Wall'} = 'Larry' ;
$h{'Walls'} = 'Brick' ;
$h{'Smith'} = 'John' ;


$key = $value = 0 ;
print "IN ORDER\n" ;
for ($st = $x->seq($key, $value, R_FIRST) ;
     $st == 0 ;
     $st = $x->seq($key, $value, R_NEXT) )

  {  print "$key -> $value\n" }

print "\nPARTIAL MATCH\n" ;

match "Wa" ;
match "A" ;
match "a" ;

undef $x ;
untie %h ;
```

Here is the output:

```
IN ORDER
Smith -> John
Wall  -> Larry
Walls -> Brick
mouse -> mickey

PARTIAL MATCH
Wa -> Wall  -> Larry
A  -> Smith -> John
a  -> mouse -> mickey
```

**DB_RECNO**

DB_RECNO provides an interface to flat text files. Both variable and fixed length records are supported.

In order to make RECNO more compatible with Perl the array offset for all RECNO arrays begins at 0 rather than 1 as in Berkeley DB.

As with normal Perl arrays, a RECNO array can be accessed using negative indexes. The index −1 refers to the last element of the array, −2 the second last, and so on. Attempting to access an element before the start of the array will raise a fatal run-time error.

### The 'bval' Option

The operation of the bval option warrants some discussion. Here is the definition of bval from the Berkeley DB 1.85 recno manual page:

```
The delimiting byte to be used to mark  the  end  of  a
record for variable-length records, and the pad charac-
ter for fixed-length records.  If no  value  is  speci-
fied,  newlines  (``\n'')  are  used to mark the end of
variable-length records and  fixed-length  records  are
padded with spaces.
```

The second sentence is wrong. In actual fact bval will only default to `"\n"` when the openinfo parameter in dbopen is NULL. If a non-NULL openinfo parameter is used at all, the value that happens to be in bval will be used. That means you always have to specify bval when making use of any of the options in the openinfo parameter. This documentation error will be fixed in the next release of Berkeley DB.

That clarifies the situation with regards Berkeley DB itself. What about **DB_File**? Well, the behavior defined in the quote above is quite useful, so **DB_File** conforms it.

That means that you can specify other options (e.g. cachesize) and still have bval default to `"\n"` for variable length records, and space for fixed length records.

### A Simple Example

Here is a simple example that uses RECNO.

```
use strict ;
use DB_File ;

my @h ;
tie @h, "DB_File", "text", O_RDWR|O_CREAT, 0640, $DB_RECNO
    or die "Cannot open file 'text': $!\n" ;

# Add a few key/value pairs to the file
$h[0] = "orange" ;
$h[1] = "blue" ;
$h[2] = "yellow" ;

# Check for existence of a key
print "Element 1 Exists with value $h[1]\n" if $h[1] ;

# use a negative index
print "The last element is $h[-1]\n" ;
print "The 2nd last element is $h[-2]\n" ;

untie @h ;
```

Here is the output from the script:

```
Element 1 Exists with value blue
The last element is yellow
The 2nd last element is blue
```

### Extra Methods

If you are using a version of Perl earlier than 5.004_57, the tied array interface is quite limited. The example script above will work, but you won't be able to use `push`, `pop`, `shift`, `unshift` etc. with the tied array.

To make the interface more useful for older versions of Perl, a number of methods are supplied with **DB_File** to simulate the missing array operations. All these methods are accessed via the object returned from the tie call.

Here are the methods:

**$X−>push(list) ;**
Pushes the elements of `list` to the end of the array.

**$value = $X−>pop ;**
Removes and returns the last element of the array.

**$X−>shift**
Removes and returns the first element of the array.

**$X−>unshift(list) ;**
Pushes the elements of `list` to the start of the array.

**$X−>length**
Returns the number of elements in the array.

### Another Example

Here is a more complete example that makes use of some of the methods described above. It also makes use of the API interface directly (see the section on *THE API INTERFACE*).

```
use strict ;
use vars qw(@h $H $file $i) ;
use DB_File ;
use Fcntl ;

$file = "text" ;

unlink $file ;

$H = tie @h, "DB_File", $file, O_RDWR|O_CREAT, 0640, $DB_RECNO
    or die "Cannot open file $file: $!\n" ;

# first create a text file to play with
$h[0] = "zero" ;
$h[1] = "one" ;
$h[2] = "two" ;
$h[3] = "three" ;
$h[4] = "four" ;
```

```
# Print the records in order.
#
# The length method is needed here because evaluating a tied
# array in a scalar context does not return the number of
# elements in the array.

print "\nORIGINAL\n" ;
foreach $i (0 .. $H->length - 1) {
    print "$i: $h[$i]\n" ;
}

# use the push & pop methods
$a = $H->pop ;
$H->push("last") ;
print "\nThe last record was [$a]\n" ;

# and the shift & unshift methods
$a = $H->shift ;
$H->unshift("first") ;
print "The first record was [$a]\n" ;

# Use the API to add a new record after record 2.
$i = 2 ;
$H->put($i, "Newbie", R_IAFTER) ;

# and a new record before record 1.
$i = 1 ;
$H->put($i, "New One", R_IBEFORE) ;

# delete record 3
$H->del(3) ;

# now print the records in reverse order
print "\nREVERSE\n" ;
for ($i = $H->length - 1 ; $i >= 0 ; -- $i)
  { print "$i: $h[$i]\n" }

# same again, but use the API functions instead
print "\nREVERSE again\n" ;
my ($s, $k, $v)  = (0, 0, 0) ;
for ($s = $H->seq($k, $v, R_LAST) ;
         $s == 0 ;
         $s = $H->seq($k, $v, R_PREV))
  { print "$k: $v\n" }

undef $H ;
untie @h ;
```

and this is what it outputs:

```
ORIGINAL
0: zero
1: one
2: two
3: three
4: four

The last record was [four]
The first record was [zero]

REVERSE
5: last
4: three
3: Newbie
2: one
1: New One
0: first

REVERSE again
5: last
4: three
3: Newbie
2: one
1: New One
0: first
```

Notes:

1. Rather than iterating through the array, `@h` like this:

   ```
   foreach $i (@h)
   ```

   it is necessary to use either this:

   ```
   foreach $i (0 .. $H->length - 1)
   ```

   or this:

   ```
   for ($a = $H->get($k, $v, R_FIRST) ;
        $a == 0 ;
        $a = $H->get($k, $v, R_NEXT) )
   ```

2. Notice that both times the `put` method was used the record index was specified using a variable, `$i`, rather than the literal value itself. This is because `put` will return the record number of the inserted line via that parameter.

**THE API INTERFACE**

As well as accessing Berkeley DB using a tied hash or array, it is also possible to make direct use of most of the API functions defined in the Berkeley DB documentation.

To do this you need to store a copy of the object returned from the tie.

```
$db = tie %hash, "DB_File", "filename" ;
```

Once you have done that, you can access the Berkeley DB API functions as **DB_File** methods directly like this:

```
$db->put($key, $value, R_NOOVERWRITE) ;
```

**Important:** If you have saved a copy of the object returned from `tie`, the underlying database file will *not* be closed until both the tied variable is untied and all copies of the saved object are destroyed.

```
use DB_File ;
$db = tie %hash, "DB_File", "filename"
    or die "Cannot tie filename: $!" ;
...
undef $db ;
untie %hash ;
```

See the section on *The untie() Gotcha* for more details.

All the functions defined in the *dbopen* manpage are available except for *close()* and *dbopen()* itself. The **DB_File** method interface to the supported functions have been implemented to mirror the way Berkeley DB works whenever possible. In particular note that:

- The methods return a status value. All return 0 on success. All return −1 to signify an error and set `$!` to the exact error code. The return code 1 generally (but not always) means that the key specified did not exist in the database.

  Other return codes are defined. See below and in the Berkeley DB documentation for details. The Berkeley DB documentation should be used as the definitive source.

- Whenever a Berkeley DB function returns data via one of its parameters, the equivalent **DB_File** method does exactly the same.

- If you are careful, it is possible to mix API calls with the tied hash/array interface in the same piece of code. Although only a few of the methods used to implement the tied interface currently make use of the cursor, you should always assume that the cursor has been changed any time the tied hash/array interface is used. As an example, this code will probably not do what you expect:

```
$X = tie %x, 'DB_File', $filename, O_RDWR|O_CREAT, 0777, $DB_BTREE
    or die "Cannot tie $filename: $!" ;

# Get the first key/value pair and set  the cursor
$X->seq($key, $value, R_FIRST) ;

# this line will modify the cursor
$count = scalar keys %x ;

# Get the second key/value pair.
# oops, it didn't, it got the last key/value pair!
$X->seq($key, $value, R_NEXT) ;
```

  The code above can be rearranged to get around the problem, like this:

```
$X = tie %x, 'DB_File', $filename, O_RDWR|O_CREAT, 0777, $DB_BTREE
    or die "Cannot tie $filename: $!" ;

# this line will modify the cursor
$count = scalar keys %x ;

# Get the first key/value pair and set  the cursor
$X->seq($key, $value, R_FIRST) ;

# Get the second key/value pair.
# worked this time.
$X->seq($key, $value, R_NEXT) ;
```

All the constants defined in the *dbopen* manpage for use in the flags parameters in the methods defined below are also available. Refer to the Berkeley DB documentation for the precise meaning of the flags values.

Below is a list of the methods available.

**$status = $X−>get($key, $value [, $flags]) ;**

Given a key (`$key`) this method reads the value associated with it from the database. The value read from the database is returned in the `$value` parameter.

If the key does not exist the method returns 1.

No flags are currently defined for this method.

**$status = $X−>put($key, $value [, $flags]) ;**

Stores the key/value pair in the database.

If you use either the R_IAFTER or R_IBEFORE flags, the `$key` parameter will have the record number of the inserted key/value pair set.

Valid flags are R_CURSOR, R_IAFTER, R_IBEFORE, R_NOOVERWRITE and R_SETCURSOR.

**$status = $X−>del($key [, $flags]) ;**

Removes all key/value pairs with key `$key` from the database.

A return code of 1 means that the requested key was not in the database.

R_CURSOR is the only valid flag at present.

**$status = $X−>fd ;**

Returns the file descriptor for the underlying database.

See the section on *Locking Databases* for an example of how to make use of the `fd` method to lock your database.

**$status = $X−>seq($key, $value, $flags) ;**

This interface allows sequential retrieval from the database. See the *dbopen* manpage for full details.

Both the `$key` and `$value` parameters will be set to the key/value pair read from the database.

The flags parameter is mandatory. The valid flag values are R_CURSOR, R_FIRST, R_LAST, R_NEXT and R_PREV.

**$status = $X−>sync([$flags]) ;**

Flushes any cached buffers to disk.

R_RECNOSYNC is the only valid flag at present.

## HINTS AND TIPS

### Locking Databases

Concurrent access of a read-write database by several parties requires them all to use some kind of locking. Here's an example of Tom's that uses the *fd* method to get the file descriptor, and then a careful *open()* to give something Perl will *flock()* for you. Run this repeatedly in the background to watch the locks granted in proper order.

```
use DB_File;

use strict;

sub LOCK_SH { 1 }
sub LOCK_EX { 2 }
sub LOCK_NB { 4 }
sub LOCK_UN { 8 }
```

```
my($oldval, $fd, $db, %db, $value, $key);

$key = shift || 'default';
$value = shift || 'magic';

$value .= " $$";

$db = tie(%db, 'DB_File', '/tmp/foo.db', O_CREAT|O_RDWR, 0644)
        || die "dbcreat /tmp/foo.db $!";
$fd = $db->fd;
print "$$: db fd is $fd\n";
open(DB_FH, "+<&=$fd") || die "dup $!";

unless (flock (DB_FH, LOCK_SH | LOCK_NB)) {
    print "$$: CONTENTION; can't read during write update!
                Waiting for read lock ($!) ....";
    unless (flock (DB_FH, LOCK_SH)) { die "flock: $!" }
}
print "$$: Read lock granted\n";

$oldval = $db{$key};
print "$$: Old value was $oldval\n";
flock(DB_FH, LOCK_UN);

unless (flock (DB_FH, LOCK_EX | LOCK_NB)) {
    print "$$: CONTENTION; must have exclusive lock!
                Waiting for write lock ($!) ....";
    unless (flock (DB_FH, LOCK_EX)) { die "flock: $!" }
}

print "$$: Write lock granted\n";
$db{$key} = $value;
$db->sync;  # to flush
sleep 10;

flock(DB_FH, LOCK_UN);
undef $db;
untie %db;
close(DB_FH);
print "$$: Updated db to $key=$value\n";
```

**Sharing Databases With C Applications**

There is no technical reason why a Berkeley DB database cannot be shared by both a Perl and a C application.

The vast majority of problems that are reported in this area boil down to the fact that C strings are NULL terminated, whilst Perl strings are not.

Here is a real example. Netscape 2.0 keeps a record of the locations you visit along with the time you last visited them in a DB_HASH database. This is usually stored in the file *~/.netscape/history.db*. The key field in the database is the location string and the value field is the time the location was last visited stored as a 4 byte binary value.

If you haven't already guessed, the location string is stored with a terminating NULL. This means you need to be careful when accessing the database.

Here is a snippet of code that is loosely based on Tom Christiansen's *ggh* script (available from your nearest CPAN archive in *authors/id/TOMC/scripts/nshist.gz*).

```
use strict ;
use DB_File ;
use Fcntl ;

use vars qw( $dotdir $HISTORY %hist_db $href $binary_time $date ) ;
$dotdir = $ENV{HOME} || $ENV{LOGNAME};

$HISTORY = "$dotdir/.netscape/history.db";

tie %hist_db, 'DB_File', $HISTORY
    or die "Cannot open $HISTORY: $!\n" ;;

# Dump the complete database
while ( ($href, $binary_time) = each %hist_db ) {

    # remove the terminating NULL
    $href =~ s/\x00$// ;

    # convert the binary time into a user friendly string
    $date = localtime unpack("V", $binary_time);
    print "$date $href\n" ;
}

# check for the existence of a specific key
# remember to add the NULL
if ( $binary_time = $hist_db{"http://mox.perl.com/\x00"} ) {
    $date = localtime unpack("V", $binary_time) ;
    print "Last visited mox.perl.com on $date\n" ;
}
else {
    print "Never visited mox.perl.com\n"
}

untie %hist_db ;
```

### The *untie()* Gotcha

If you make use of the Berkeley DB API, it is *very* strongly recommended that you read the section on *The untie Gotcha* in the *perltie* manpage.

Even if you don't currently make use of the API interface, it is still worth reading it.

Here is an example which illustrates the problem from a **DB_File** perspective:

```
use DB_File ;
use Fcntl ;

my %x ;
my $X ;

$X = tie %x, 'DB_File', 'tst.fil' , O_RDWR|O_TRUNC
    or die "Cannot tie first time: $!" ;
```

```
$x{123} = 456 ;

untie %x ;

tie %x, 'DB_File', 'tst.fil' , O_RDWR|O_CREAT
    or die "Cannot tie second time: $!" ;

untie %x ;
```

When run, the script will produce this error message:

```
Cannot tie second time: Invalid argument at bad.file line 14.
```

Although the error message above refers to the second *tie()* statement in the script, the source of the problem is really with the *untie()* statement that precedes it.

Having read the *perltie* manpage you will probably have already guessed that the error is caused by the extra copy of the tied object stored in `$X`. If you haven't, then the problem boils down to the fact that the **DB_File** destructor, DESTROY, will not be called until *all* references to the tied object are destroyed. Both the tied variable, `%x`, and `$X` above hold a reference to the object. The call to *untie()* will destroy the first, but `$X` still holds a valid reference, so the destructor will not get called and the database file *tst.fil* will remain open. The fact that Berkeley DB then reports the attempt to open a database that is alreday open via the catch-all "Invalid argument" doesn't help.

If you run the script with the `-w` flag the error message becomes:

```
untie attempted while 1 inner references still exist at bad.file line 12.
Cannot tie second time: Invalid argument at bad.file line 14.
```

which pinpoints the real problem. Finally the script can now be modified to fix the original problem by destroying the API object before the untie:

```
...
$x{123} = 456 ;

undef $X ;
untie %x ;

$X = tie %x, 'DB_File', 'tst.fil' , O_RDWR|O_CREAT
...
```

## COMMON QUESTIONS

### Why is there Perl source in my database?

If you look at the contents of a database file created by DB_File, there can sometimes be part of a Perl script included in it.

This happens because Berkeley DB uses dynamic memory to allocate buffers which will subsequently be written to the database file. Being dynamic, the memory could have been used for anything before DB malloced it. As Berkeley DB doesn't clear the memory once it has been allocated, the unused portions will contain random junk. In the case where a Perl script gets written to the database, the random junk will correspond to an area of dynamic memory that happened to be used during the compilation of the script.

Unless you don't like the possibility of there being part of your Perl scripts embedded in a database file, this is nothing to worry about.

### How do I store complex data structures with DB_File?

Although **DB_File** cannot do this directly, there is a module which can layer transparently over **DB_File** to accomplish this feat.

Check out the MLDBM module, available on CPAN in the directory *modules/by-module/MLDBM*.

### What does "Invalid Argument" mean?

You will get this error message when one of the parameters in the `tie` call is wrong. Unfortunately there are quite a few parameters to get wrong, so it can be difficult to figure out which one it is.

Here are a couple of possibilities:

1. Attempting to reopen a database without closing it.
2. Using the O_WRONLY flag.

### What does "Bareword 'DB_File' not allowed" mean?

You will encounter this particular error message when you have the `strict 'subs'` pragma (or the full strict pragma) in your script. Consider this script:

```
use strict ;
use DB_File ;
use vars qw(%x) ;
tie %x, DB_File, "filename" ;
```

Running it produces the error in question:

```
Bareword "DB_File" not allowed while "strict subs" in use
```

To get around the error, place the word `DB_File` in either single or double quotes, like this:

```
tie %x, "DB_File", "filename" ;
```

Although it might seem like a real pain, it is really worth the effort of having a `use strict` in all your scripts.

## HISTORY

Moved to the Changes file.

## BUGS

Some older versions of Berkeley DB had problems with fixed length records using the RECNO file format. This problem has been fixed since version 1.85 of Berkeley DB.

I am sure there are bugs in the code. If you do find any, or can suggest any enhancements, I would welcome your comments.

## AVAILABILITY

**DB_File** comes with the standard Perl source distribution. Look in the directory *ext/DB_File*. Given the amount of time between releases of Perl the version that ships with Perl is quite likely to be out of date, so the most recent version can always be found on CPAN (see the `CPAN` entry in the *perlmod* manpage for details), in the directory *modules/by-module/DB_File*.

This version of **DB_File** will work with either version 1.x or 2.x of Berkeley DB, but is limited to the functionality provided by version 1.

The official web site for Berkeley DB is *http://www.sleepycat.com/db*. The ftp equivalent is *ftp.sleepycat.com:/pub*. Both versions 1 and 2 of Berkeley DB are available there.

Alternatively, Berkeley DB version 1 is available at your nearest CPAN archive in *src/misc/db.1.85.tar.gz*.

If you are running IRIX, then get Berkeley DB version 1 from *http://reality.sgi.com/ariel*. It has the patches necessary to compile properly on IRIX 5.3.

**COPYRIGHT**



Although **DB_File** is covered by the Perl license, the library it makes use of, namely Berkeley DB, is not. Berkeley DB has its own copyright and its own license. Please take the time to read it.

Here are are few words taken from the Berkeley DB FAQ (at http://www.sleepycat.com) regarding the license:

```
Do I have to license DB to use it in Perl scripts?

No. The Berkeley DB license requires that software that uses
Berkeley DB be freely redistributable. In the case of Perl, that
software is Perl, and not your scripts. Any Perl scripts that you
write are your property, including scripts that make use of
Berkeley DB. Neither the Perl license nor the Berkeley DB license
place any restriction on what you may do with them.
```

If you are in any doubt about the license situation, contact either the Berkeley DB authors or the author of DB_File. See the section on *AUTHOR* for details.

**SEE ALSO**

the *perl(1)* manpage, the *dbopen(3)* manpage, the *hash(3)* manpage, the *recno(3)* manpage, the *btree(3)* manpage

**AUTHOR**

The DB_File interface was written by Paul Marquess <pmarquess@bfsec.bt.co.uk>. Questions about the DB system itself may be addressed to <db@sleepycat.com<gt>.

**NAME**

Devel::SelfStubber − generate stubs for a SelfLoading module

**SYNOPSIS**

To generate just the stubs:

```
use Devel::SelfStubber;
Devel::SelfStubber−>stub('MODULENAME','MY_LIB_DIR');
```

or to generate the whole module with stubs inserted correctly

```
use Devel::SelfStubber;
$Devel::SelfStubber::JUST_STUBS=0;
Devel::SelfStubber−>stub('MODULENAME','MY_LIB_DIR');
```

MODULENAME is the Perl module name, e.g. Devel::SelfStubber, NOT 'Devel/SelfStubber' or 'Devel/SelfStubber.pm'.

MY_LIB_DIR defaults to '.' if not present.

**DESCRIPTION**

Devel::SelfStubber prints the stubs you need to put in the module before the __DATA__ token (or you can get it to print the entire module with stubs correctly placed). The stubs ensure that if a method is called, it will get loaded. They are needed specifically for inherited autoloaded methods.

This is best explained using the following example:

Assume four classes, A,B,C & D.

A is the root class, B is a subclass of A, C is a subclass of B, and D is another subclass of A.

```
        A
       / \
      B   D
     /
    C
```

If D calls an autoloaded method 'foo' which is defined in class A, then the method is loaded into class A, then executed. If C then calls method 'foo', and that method was reimplemented in class B, but set to be autoloaded, then the lookup mechanism never gets to the AUTOLOAD mechanism in B because it first finds the method already loaded in A, and so erroneously uses that. If the method foo had been stubbed in B, then the lookup mechanism would have found the stub, and correctly loaded and used the sub from B.

So, for classes and subclasses to have inheritance correctly work with autoloading, you need to ensure stubs are loaded.

The SelfLoader can load stubs automatically at module initialization with the statement 'SelfLoader->*load_stubs()*';, but you may wish to avoid having the stub loading overhead associated with your initialization (though note that the SelfLoader::load_stubs method will be called sooner or later − at latest when the first sub is being autoloaded). In this case, you can put the sub stubs before the __DATA__ token. This can be done manually, but this module allows automatic generation of the stubs.

By default it just prints the stubs, but you can set the global `$Devel::SelfStubber::JUST_STUBS` to 0 and it will print out the entire module with the stubs positioned correctly.

At the very least, this is useful to see what the SelfLoader thinks are stubs − in order to ensure future versions of the SelfStubber remain in step with the SelfLoader, the SelfStubber actually uses the SelfLoader to determine which stubs are needed.

## NAME

DirHandle − supply object methods for directory handles

## SYNOPSIS

```
use DirHandle;
$d = new DirHandle ".";
if (defined $d) {
    while (defined($_ = $d->read)) { something($_); }
    $d->rewind;
    while (defined($_ = $d->read)) { something_else($_); }
    undef $d;
}
```

## DESCRIPTION

The `DirHandle` method provide an alternative interface to the *opendir()*, *closedir()*, *readdir()*, and *rewinddir()* functions.

The only objective benefit to using `DirHandle` is that it avoids namespace pollution by creating globs to hold directory handles.

## NAME

DynaLoader − Dynamically load C libraries into Perl code

*dl_error()*, *dl_findfile()*, *dl_expandspec()*, *dl_load_file()*, *dl_find_symbol()*, *dl_find_symbol_anywhere()*, *dl_undef_symbols()*, *dl_install_xsub()*, *dl_load_flags()*, *bootstrap()* − routines used by DynaLoader modules

## SYNOPSIS

```
package YourPackage;
require DynaLoader;
@ISA = qw(... DynaLoader ...);
bootstrap YourPackage;

# optional method for 'global' loading
sub dl_load_flags { 0x01 }
```

## DESCRIPTION

This document defines a standard generic interface to the dynamic linking mechanisms available on many platforms. Its primary purpose is to implement automatic dynamic loading of Perl modules.

This document serves as both a specification for anyone wishing to implement the DynaLoader for a new platform and as a guide for anyone wishing to use the DynaLoader directly in an application.

The DynaLoader is designed to be a very simple high-level interface that is sufficiently general to cover the requirements of SunOS, HP−UX, NeXT, Linux, VMS and other platforms.

It is also hoped that the interface will cover the needs of OS/2, NT etc and also allow pseudo-dynamic linking (using `ld −A` at runtime).

It must be stressed that the DynaLoader, by itself, is practically useless for accessing non-Perl libraries because it provides almost no Perl-to-C 'glue'. There is, for example, no mechanism for calling a C library function or supplying arguments. A C::DynaLib module is available from CPAN sites which performs that function for some common system types.

DynaLoader Interface Summary

```
@dl_library_path
@dl_resolve_using
@dl_require_symbols
$dl_debug
@dl_librefs
@dl_modules
                                                  Implemented in:
bootstrap($modulename)                                 Perl
@filepaths = dl_findfile(@names)                       Perl
$flags = $modulename->dl_load_flags                    Perl
$symref  = dl_find_symbol_anywhere($symbol)            Perl

$libref  = dl_load_file($filename, $flags)             C
$symref  = dl_find_symbol($libref, $symbol)            C
@symbols = dl_undef_symbols()                          C
dl_install_xsub($name, $symref [, $filename])          C
$message = dl_error                                    C
```

@dl_library_path

The standard/default list of directories in which *dl_findfile()* will search for libraries etc. Directories are searched in order: `$dl_library_path`[0], [1], ... etc

@dl_library_path is initialised to hold the list of 'normal' directories (*/usr/lib*, etc) determined by

**Configure** (`$Config{'libpth'}`). This should ensure portability across a wide range of platforms.

@dl_library_path should also be initialised with any other directories that can be determined from the environment at runtime (such as LD_LIBRARY_PATH for SunOS).

After initialisation `@dl_library_path` can be manipulated by an application using push and unshift before calling *dl_findfile()*. Unshift can be used to add directories to the front of the search order either to save search time or to override libraries with the same name in the 'normal' directories.

The load function that *dl_load_file()* calls may require an absolute pathname. The *dl_findfile()* function and `@dl_library_path` can be used to search for and return the absolute pathname for the library/object that you wish to load.

@dl_resolve_using
A list of additional libraries or other shared objects which can be used to resolve any undefined symbols that might be generated by a later call to *load_file()*.

This is only required on some platforms which do not handle dependent libraries automatically. For example the Socket Perl extension library (*auto/Socket/Socket.so*) contains references to many socket functions which need to be resolved when it's loaded. Most platforms will automatically know where to find the 'dependent' library (e.g., */usr/lib/libsocket.so*). A few platforms need to be told the location of the dependent library explicitly. Use `@dl_resolve_using` for this.

Example usage:

```
@dl_resolve_using = dl_findfile('-lsocket');
```

@dl_require_symbols
A list of one or more symbol names that are in the library/object file to be dynamically loaded. This is only required on some platforms.

@dl_librefs
An array of the handles returned by successful calls to *dl_load_file()*, made by bootstrap, in the order in which they were loaded. Can be used with *dl_find_symbol()* to look for a symbol in any of the loaded files.

@dl_modules
An array of module (package) names that have been bootstrap'ed.

dl_error()
Syntax:

```
$message = dl_error();
```

Error message text from the last failed DynaLoader function. Note that, similar to errno in unix, a successful function call does not reset this message.

Implementations should detect the error as soon as it occurs in any of the other functions and save the corresponding message for later retrieval. This will avoid problems on some platforms (such as SunOS) where the error message is very temporary (e.g., *dlerror()*).

$dl_debug
Internal debugging messages are enabled when `$dl_debug` is set true. Currently setting `$dl_debug` only affects the Perl side of the DynaLoader. These messages should help an application developer to resolve any DynaLoader usage problems.

$dl_debug is set to `$ENV{'PERL_DL_DEBUG'}` if defined.

For the DynaLoader developer/porter there is a similar debugging variable added to the C code (see dlutils.c) and enabled if Perl was built with the **−DDEBUGGING** flag. This can also be set via the PERL_DL_DEBUG environment variable. Set to 1 for minimal information or higher for more.

dl_findfile()

Syntax:

```
@filepaths = dl_findfile(@names)
```

Determine the full paths (including file suffix) of one or more loadable files given their generic names and optionally one or more directories. Searches directories in `@dl_library_path` by default and returns an empty list if no files were found.

Names can be specified in a variety of platform independent forms. Any names in the form **–lname** are converted into *libname.**, where *.** is an appropriate suffix for the platform.

If a name does not already have a suitable prefix and/or suffix then the corresponding file will be searched for by trying combinations of prefix and suffix appropriate to the platform: "$name.o", "lib$name.*" and "$name".

If any directories are included in `@names` they are searched before `@dl_library_path`. Directories may be specified as **–Ldir**. Any other names are treated as filenames to be searched for.

Using arguments of the form `-Ldir` and `-lname` is recommended.

Example:

```
@dl_resolve_using = dl_findfile(qw(-L/usr/5lib -lposix));
```

dl_expandspec()

Syntax:

```
$filepath = dl_expandspec($spec)
```

Some unusual systems, such as VMS, require special filename handling in order to deal with symbolic names for files (i.e., VMS's Logical Names).

To support these systems a *dl_expandspec()* function can be implemented either in the *dl_*.xs* file or code can be added to the autoloadable *dl_expandspec()* function in *DynaLoader.pm*. See *DynaLoader.pm* for more information.

dl_load_file()

Syntax:

```
$libref = dl_load_file($filename, $flags)
```

Dynamically load `$filename`, which must be the path to a shared object or library. An opaque 'library reference' is returned as a handle for the loaded object. Returns undef on error.

The `$flags` argument to alters dl_load_file behaviour. Assigned bits:

```
 0x01  make symbols available for linking later dl_load_file's.
       (only known to work on Solaris 2 using dlopen(RTLD_GLOBAL))
       (ignored under VMS; this is a normal part of image linking)
```

(On systems that provide a handle for the loaded object such as SunOS and HPUX, `$libref` will be that handle. On other systems `$libref` will typically be `$filename` or a pointer to a buffer containing `$filename`. The application should not examine or alter `$libref` in any way.)

This is the function that does the real work. It should use the current values of `@dl_require_symbols` and `@dl_resolve_using` if required.

```
SunOS: dlopen($filename)
HP-UX: shl_load($filename)
Linux: dld_create_reference(@dl_require_symbols); dld_link($filename)
NeXT:  rld_load($filename, @dl_resolve_using)
VMS:   lib$find_image_symbol($filename,$dl_require_symbols[0])
```

(The *dlopen()* function is also used by Solaris and some versions of Linux, and is a common choice when providing a ''wrapper'' on other mechanisms as is done in the OS/2 port.)

dl_loadflags()
Syntax:

```
$flags = dl_loadflags $modulename;
```

Designed to be a method call, and to be overridden by a derived class (i.e. a class which has DynaLoader in its `@ISA`). The definition in DynaLoader itself returns 0, which produces standard behavior from *dl_load_file()*.

dl_find_symbol()
Syntax:

```
$symref = dl_find_symbol($libref, $symbol)
```

Return the address of the symbol `$symbol` or `undef` if not found. If the target system has separate functions to search for symbols of different types then *dl_find_symbol()* should search for function symbols first and then other types.

The exact manner in which the address is returned in `$symref` is not currently defined. The only initial requirement is that `$symref` can be passed to, and understood by, *dl_install_xsub()*.

```
SunOS: dlsym($libref, $symbol)
HP-UX: shl_findsym($libref, $symbol)
Linux: dld_get_func($symbol) and/or dld_get_symbol($symbol)
NeXT:  rld_lookup("_$symbol")
VMS:   lib$find_image_symbol($libref,$symbol)
```

dl_find_symbol_anywhere()
Syntax:

```
$symref = dl_find_symbol_anywhere($symbol)
```

Applies *dl_find_symbol()* to the members of `@dl_librefs` and returns the first match found.

dl_undef_symbols()
Example

```
@symbols = dl_undef_symbols()
```

Return a list of symbol names which remain undefined after *load_file()*. Returns `()` if not known. Don't worry if your platform does not provide a mechanism for this. Most do not need it and hence do not provide it, they just return an empty list.

dl_install_xsub()
Syntax:

```
dl_install_xsub($perl_name, $symref [, $filename])
```

Create a new Perl external subroutine named `$perl_name` using `$symref` as a pointer to the function which implements the routine. This is simply a direct call to *newXSUB()*. Returns a reference to the installed function.

The `$filename` parameter is used by Perl to identify the source file for the function if required by *die()*, *caller()* or the debugger. If `$filename` is not defined then "DynaLoader" will be used.

bootstrap()

Syntax:

*bootstrap* ($module)

This is the normal entry point for automatic dynamic loading in Perl.

It performs the following actions:

- locates an auto/$module directory by searching `@INC`
- uses *dl_findfile()* to determine the filename to load
- sets `@dl_require_symbols` to `("boot_$module")`
- executes an *auto/$module/$module.bs* file if it exists (typically used to add to `@dl_resolve_using` any files which are required to load the module on the current platform)
- calls *dl_load_flags()* to determine how to load the file.
- calls *dl_load_file()* to load the file
- calls *dl_undef_symbols()* and warns if any symbols are undefined
- calls *dl_find_symbol()* for "boot_$module"
- calls *dl_install_xsub()* to install it as "${module}::bootstrap"
- calls &{"${module}::bootstrap"} to bootstrap the module (actually it uses the function reference returned by dl_install_xsub for speed)

**AUTHOR**

Tim Bunce, 11 August 1994.

This interface is based on the work and comments of (in no particular order): Larry Wall, Robert Sanders, Dean Roehrich, Jeff Okamoto, Anno Siegel, Thomas Neumann, Paul Marquess, Charles Bailey, myself and others.

Larry Wall designed the elegant inherited bootstrap mechanism and implemented the first Perl 5 dynamic loader using it.

Solaris global loading added by Nick Ing-Simmons with design/coding assistance from Tim Bunce, January 1996.

**NAME**

English − use nice English (or awk) names for ugly punctuation variables

**SYNOPSIS**

```
use English;
...
if ($ERRNO =~ /denied/) { ... }
```

**DESCRIPTION**

This module provides aliases for the built-in variables whose names no one seems to like to read. Variables with side-effects which get triggered just by accessing them (like `$0`) will still be affected.

For those variables that have an **awk** version, both long and short English alternatives are provided. For example, the `$/` variable can be referred to either `$RS` or `$INPUT_RECORD_SEPARATOR` if you are using the English module.

See the *perlvar* manpage for a complete list of these.

## NAME

Env − perl module that imports environment variables

## SYNOPSIS

```
use Env;
use Env qw(PATH HOME TERM);
```

## DESCRIPTION

Perl maintains environment variables in a pseudo-hash named `%ENV`. For when this access method is inconvenient, the Perl module `Env` allows environment variables to be treated as simple variables.

The *Env::import()* function ties environment variables with suitable names to global Perl variables with the same names. By default it does so with all existing environment variables (`keys %ENV`). If the import function receives arguments, it takes them to be a list of environment variables to tie; it's okay if they don't yet exist.

After an environment variable is tied, merely use it like a normal variable. You may access its value

```
@path = split(/:/, $PATH);
```

or modify it

```
$PATH .= ":.";
```

however you'd like. To remove a tied environment variable from the environment, assign it the undefined value

```
undef $PATH;
```

## AUTHOR

Chip Salzenberg <*chip@fin.uucp*>

## NAME

Errno – System errno constants

## SYNOPSIS

```
use Errno qw(EINTR EIO :POSIX);
```

## DESCRIPTION

`Errno` defines and conditionally exports all the error constants defined in your system `errno.h` include file. It has a single export tag, `:POSIX`, which will export all POSIX defined error numbers.

`Errno` also makes `%!` magic such that each element of `%!` has a non-zero value only if `$!` is set to that value, eg

```
use Errno;

unless (open(FH, "/fangorn/spouse")) {
    if ($!{ENOENT}) {
        warn "Get a wife!\n";
    } else {
        warn "This path is barred: $!";
    }
}
```

## AUTHOR

Graham Barr <gbarr@pobox.com>

## COPYRIGHT

## NAME

Exporter – Implements default import method for modules

## SYNOPSIS

In module ModuleName.pm:

```
package ModuleName;
require Exporter;
@ISA = qw(Exporter);

@EXPORT = qw(...);            # symbols to export by default
@EXPORT_OK = qw(...);         # symbols to export on request
%EXPORT_TAGS = tag => [...];  # define names for sets of symbols
```

In other files which wish to use ModuleName:

```
use ModuleName;               # import default symbols into my package

use ModuleName qw(...);       # import listed symbols into my package

use ModuleName ();            # do not import any symbols
```

## DESCRIPTION

The Exporter module implements a default `import` method which many modules choose to inherit rather than implement their own.

Perl automatically calls the `import` method when processing a `use` statement for a module. Modules and `use` are documented in the *perlfunc* manpage and the *perlmod* manpage. Understanding the concept of modules and how the `use` statement operates is important to understanding the Exporter.

### Selecting What To Export

Do **not** export method names!

Do **not** export anything else by default without a good reason!

Exports pollute the namespace of the module user. If you must export try to use `@EXPORT_OK` in preference to `@EXPORT` and avoid short or common symbol names to reduce the risk of name clashes.

Generally anything not exported is still accessible from outside the module using the ModuleName::item_name (or `$blessed_ref->`method) syntax. By convention you can use a leading underscore on names to informally indicate that they are 'internal' and not for public use.

(It is actually possible to get private functions by saying:

```
my $subref = sub { ... };
&$subref;
```

But there's no way to call that directly as a method, since a method must have a name in the symbol table.)

As a general rule, if the module is trying to be object oriented then export nothing. If it's just a collection of functions then `@EXPORT_OK` anything but use `@EXPORT` with caution.

Other module design guidelines can be found in the *perlmod* manpage.

### Specialised Import Lists

If the first entry in an import list begins with !, : or / then the list is treated as a series of specifications which either add to or delete from the list of names to import. They are processed left to right. Specifications are in the form:

```
[!]name         This name only
[!]:DEFAULT     All names in @EXPORT
[!]:tag         All names in $EXPORT_TAGS{tag} anonymous list
[!]/pattern/    All names in @EXPORT and @EXPORT_OK which match
```

A leading ! indicates that matching names should be deleted from the list of names to import. If the first specification is a deletion it is treated as though preceded by :DEFAULT. If you just want to import extra names in addition to the default set you will still need to include :DEFAULT explicitly.

e.g., Module.pm defines:

```
@EXPORT      = qw(A1 A2 A3 A4 A5);
@EXPORT_OK   = qw(B1 B2 B3 B4 B5);
%EXPORT_TAGS = (T1 => [qw(A1 A2 B1 B2)], T2 => [qw(A1 A2 B3 B4)]);

Note that you cannot use tags in @EXPORT or @EXPORT_OK.
Names in EXPORT_TAGS must also appear in @EXPORT or @EXPORT_OK.
```

An application using Module can say something like:

```
use Module qw(:DEFAULT :T2 !B3 A3);
```

Other examples include:

```
use Socket qw(!/^[AP]F_/ !SOMAXCONN !SOL_SOCKET);
use POSIX  qw(:errno_h :termios_h !TCSADRAIN !/^EXIT/);
```

Remember that most patterns (using //) will need to be anchored with a leading ^, e.g., `/^EXIT/` rather than `/EXIT/`.

You can say `BEGIN { $Exporter::Verbose=1 }` to see how the specifications are being processed and what is actually being imported into modules.

**Exporting without using Export's import method**

Exporter has a special method, 'export_to_level' which is used in situations where you can't directly call Export's import method. The export_to_level method looks like:

MyPackage->*export_to_level*($where_to_export, `@what_to_export`);

where `$where_to_export` is an integer telling how far up the calling stack to export your symbols, and `@what_to_export` is an array telling what symbols *to* export (usually this is `@_`).

For example, suppose that you have a module, A, which already has an import function:

package A;

@ISA = *qw*(Exporter); `@EXPORT_OK` = qw ($b);

sub import {
    `$A::b` = 1;     # not a very useful import method }

and you want to Export symbol `$A::b` back to the module that called package A. Since Exporter relies on the import method to work, via inheritance, as it stands *Exporter::import()* will never get called. Instead, say the following:

package A; `@ISA` = *qw*(Exporter); `@EXPORT_OK` = qw ($b);

sub import {
    `$A::b` = 1;
  A–>*export_to_level*(1, `@_`); }

This will export the symbols one level 'above' the current package – ie: to the program or module that used package A.

Note: Be careful not to modify '@_' at all before you call export_to_level – or people using your package will get very unexplained results!

### Module Version Checking

The Exporter module will convert an attempt to import a number from a module into a call to `$module_name->`*require_version* ($value). This can be used to validate that the version of the module being used is greater than or equal to the required version.

The Exporter module supplies a default require_version method which checks the value of `$VERSION` in the exporting module.

Since the default require_version method treats the `$VERSION` number as a simple numeric value it will regard version 1.10 as lower than 1.9. For this reason it is strongly recommended that you use numbers with at least two decimal places, e.g., 1.09.

### Managing Unknown Symbols

In some situations you may want to prevent certain symbols from being exported. Typically this applies to extensions which have functions or constants that may not exist on some systems.

The names of any symbols that cannot be exported should be listed in the `@EXPORT_FAIL` array.

If a module attempts to import any of these symbols the Exporter will give the module an opportunity to handle the situation before generating an error. The Exporter will call an export_fail method with a list of the failed symbols:

```
@failed_symbols = $module_name->export_fail(@failed_symbols);
```

If the export_fail method returns an empty list then no error is recorded and all the requested symbols are exported. If the returned list is not empty then an error is generated for each symbol and the export fails. The Exporter provides a default export_fail method which simply returns the list unchanged.

Uses for the export_fail method include giving better error messages for some symbols and performing lazy architectural checks (put more symbols into `@EXPORT_FAIL` by default and then take them out if someone actually tries to use them and an expensive check shows that they are usable on that platform).

### Tag Handling Utility Functions

Since the symbols listed within `%EXPORT_TAGS` must also appear in either `@EXPORT` or `@EXPORT_OK`, two utility functions are provided which allow you to easily add tagged sets of symbols to `@EXPORT` or `@EXPORT_OK`:

```
%EXPORT_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);

Exporter::export_tags('foo');     # add aa, bb and cc to @EXPORT
Exporter::export_ok_tags('bar');  # add aa, cc and dd to @EXPORT_OK
```

Any names which are not tags are added to `@EXPORT` or `@EXPORT_OK` unchanged but will trigger a warning (with `-w`) to avoid misspelt tags names being silently added to `@EXPORT` or `@EXPORT_OK`. Future versions may make this a fatal error.

## NAME

ExtUtils::Command – utilities to replace common UNIX commands in Makefiles etc.

## SYNOPSIS

```
perl -MExtUtils::Command -e cat files... > destination
perl -MExtUtils::Command -e mv source... destination
perl -MExtUtils::Command -e cp source... destination
perl -MExtUtils::Command -e touch files...
perl -MExtUtils::Command -e rm_f file...
perl -MExtUtils::Command -e rm_rf directories...
perl -MExtUtils::Command -e mkpath directories...
perl -MExtUtils::Command -e eqtime source destination
perl -MExtUtils::Command -e chmod mode files...
perl -MExtUtils::Command -e test_f file
```

## DESCRIPTION

The module is used in Win32 port to replace common UNIX commands. Most commands are wrapers on generic modules File::Path and File::Basename.

cat Concatenates all files mentioned on command line to STDOUT.

eqtime src dst
: Sets modified time of dst to that of src

rm_f files....
: Removes directories – recursively (even if readonly)

rm_f files....
: Removes files (even if readonly)

touch files ...
: Makes files exist, with current timestamp

mv source... destination
: Moves source to destination. Multiple sources are allowed if destination is an existing directory.

cp source... destination
: Copies source to destination. Multiple sources are allowed if destination is an existing directory.

chmod mode files...
: Sets UNIX like permissions 'mode' on all the files.

mkpath directory...
: Creates directory, including any parent directories.

test_f file
: Tests if a file exists

## BUGS

Should probably be Auto/Self loaded.

## SEE ALSO

ExtUtils::MakeMaker, ExtUtils::MM_Unix, ExtUtils::MM_Win32

## AUTHOR

Nick Ing-Simmons <*nick@ni-s.u-net.com*>.

## NAME

ExtUtils::Embed – Utilities for embedding Perl in C/C++ applications

## SYNOPSIS

```
perl −MExtUtils::Embed −e xsinit
perl −MExtUtils::Embed −e ldopts
```

## DESCRIPTION

ExtUtils::Embed provides utility functions for embedding a Perl interpreter and extensions in your C/C++ applications. Typically, an application **Makefile** will invoke ExtUtils::Embed functions while building your application.

### @EXPORT

ExtUtils::Embed exports the following functions:

*xsinit()*, *ldopts()*, *ccopts()*, *perl_inc()*, *ccflags()*, *ccdlflags()*, *xsi_header()*, *xsi_protos()*, *xsi_body()*

## FUNCTIONS

xsinit()

Generate C/C++ code for the XS initializer function.

When invoked as `` `perl −MExtUtils::Embed −e xsinit −−` `` the following options are recognized:

**−o** <output filename> (Defaults to **perlxsi.c**)

**−o STDOUT** will print to STDOUT.

**−std** (Write code for extensions that are linked with the current Perl.)

Any additional arguments are expected to be names of modules to generate code for.

When invoked with parameters the following are accepted and optional:

`xsinit($filename,$std,[@modules])`

Where,

**$filename** is equivalent to the **−o** option.

**$std** is boolean, equivalent to the **−std** option.

**[@modules]** is an array ref, same as additional arguments mentioned above.

Examples

```
perl −MExtUtils::Embed −e xsinit −− −o xsinit.c Socket
```

This will generate code with an **xs_init** function that glues the perl **Socket::bootstrap** function to the C **boot_Socket** function and writes it to a file named "xsinit.c".

Note that **DynaLoader** is a special case where it must call **boot_DynaLoader** directly.

```
perl −MExtUtils::Embed −e xsinit
```

This will generate code for linking with **DynaLoader** and each static extension found in **$Config{static_ext}**. The code is written to the default file name **perlxsi.c**.

```
perl −MExtUtils::Embed −e xsinit −− −o xsinit.c −std DBI DBD::Oracle
```

Here, code is written for all the currently linked extensions along with code for **DBI** and **DBD::Oracle**.

If you have a working **DynaLoader** then there is rarely any need to statically link in any other extensions.

ldopts()

Output arguments for linking the Perl library and extensions to your application.

When invoked as `perl –MExtUtils::Embed –e ldopts --` the following options are recognized:

**–std**

Output arguments for linking the Perl library and any extensions linked with the current Perl.

**–I** <path1:path2>

Search path for ModuleName.a archives. Default path is **@INC**. Library archives are expected to be found as **/some/path/auto/ModuleName/ModuleName.a** For example, when looking for **Socket.a** relative to a search path, we should find **auto/Socket/Socket.a**

When looking for **DBD::Oracle** relative to a search path, we should find **auto/DBD/Oracle/Oracle.a**

Keep in mind, you can always supply **/my/own/path/ModuleName.a** as an additional linker argument.

**--** <list of linker args>

Additional linker arguments to be considered.

Any additional arguments found before the **--** token are expected to be names of modules to generate code for.

When invoked with parameters the following are accepted and optional:

`ldopts($std,[@modules],[@link_args],$path)`

Where,

**$std** is boolean, equivalent to the **–std** option.

**[@modules]** is equivalent to additional arguments found before the **--** token.

**[@link_args]** is equivalent to arguments found after the **--** token.

**$path** is equivalent to the **–I** option.

In addition, when ldopts is called with parameters, it will return the argument string rather than print it to STDOUT.

Examples

```
perl –MExtUtils::Embed –e ldopts
```

This will print arguments for linking with **libperl.a**, **DynaLoader** and extensions found in **$Config{static_ext}**. This includes libraries found in **$Config{libs}** and the first ModuleName.a library for each extension that is found by searching **@INC** or the path specifed by the **–I** option. In addition, when ModuleName.a is found, additional linker arguments are picked up from the **extralibs.ld** file in the same directory.

```
perl –MExtUtils::Embed –e ldopts -- –std Socket
```

This will do the same as the above example, along with printing additional arguments for linking with the **Socket** extension.

```
perl –MExtUtils::Embed –e ldopts -- DynaLoader
```

This will print arguments for linking with just the **DynaLoader** extension and **libperl.a**.

```
perl –MExtUtils::Embed –e ldopts -- –std Msql -- –L/usr/msql/lib –lmsql
```

Any arguments after the second '--' token are additional linker arguments that will be examined for potential conflict. If there is no conflict, the additional arguments will be part of the output.

perl_inc()
For including perl header files this function simply prints:

```
-I$Config{archlibexp}/CORE
```

So, rather than having to say:

```
perl -MConfig -e 'print "-I$Config{archlibexp}/CORE"'
```

Just say:

```
perl -MExtUtils::Embed -e perl_inc
```

ccflags(), ccdlflags()
These functions simply print `$Config`{ccflags} and `$Config`{ccdlflags}

ccopts()
This function combines *perl_inc()*, *ccflags()* and *ccdlflags()* into one.

xsi_header()
This function simply returns a string defining the same **EXTERN_C** macro as **perlmain.c** along with #including **perl.h** and **EXTERN.h**.

xsi_protos(@modules)
This function returns a string of **boot_$ModuleName** prototypes for each `@modules`.

xsi_body(@modules)
This function returns a string of calls to **newXS()** that glue the module **bootstrap** function to **boot_ModuleName** for each `@modules`.

**xsinit()** uses the xsi_* functions to generate most of it's code.

**EXAMPLES**

For examples on how to use **ExtUtils::Embed** for building C/C++ applications with embedded perl, see the eg/ directory and the *perlembed* manpage.

**SEE ALSO**

the *perlembed* manpage

**AUTHOR**

Doug MacEachern <*dougm@osf.org*>

Based on ideas from Tim Bunce <*Tim.Bunce@ig.co.uk*> and **minimod.pl** by Andreas Koenig <*k@anna.in-berlin.de*> and Tim Bunce.

## NAME

ExtUtils::Install − install files from here to there

## SYNOPSIS

**use ExtUtils::Install;**

**install($hashref,$verbose,$nonono);**

**uninstall($packlistfile,$verbose,$nonono);**

**pm_to_blib($hashref);**

## DESCRIPTION

Both *install()* and *uninstall()* are specific to the way ExtUtils::MakeMaker handles the installation and deinstallation of perl modules. They are not designed as general purpose tools.

*install()* takes three arguments. A reference to a hash, a verbose switch and a don't-really-do-it switch. The hash ref contains a mapping of directories: each key/value pair is a combination of directories to be copied. Key is a directory to copy from, value is a directory to copy to. The whole tree below the ''from'' directory will be copied preserving timestamps and permissions.

There are two keys with a special meaning in the hash: ''read'' and ''write''. After the copying is done, install will write the list of target files to the file named by `$hashref->{write}`. If there is another file named by `$hashref->{read}`, the contents of this file will be merged into the written file. The read and the written file may be identical, but on AFS it is quite likely, people are installing to a different directory than the one where the files later appear.

*install_default()* takes one or less arguments. If no arguments are specified, it takes `$ARGV`[0] as if it was specified as an argument. The argument is the value of MakeMaker's `FULLEXT` key, like *Tk/Canvas*. This function calls *install()* with the same arguments as the defaults the MakeMaker would use.

The argumement-less form is convenient for install scripts like

```
perl −MExtUtils::Install −e install_default Tk/Canvas
```

Assuming this command is executed in a directory with populated *blib* directory, it will proceed as if the *blib* was build by MakeMaker on this machine. This is useful for binary distributions.

*uninstall()* takes as first argument a file containing filenames to be unlinked. The second argument is a verbose switch, the third is a no-don't-really-do-it-now switch.

*pm_to_blib()* takes a hashref as the first argument and copies all keys of the hash to the corresponding values efficiently. Filenames with the extension pm are autosplit. Second argument is the autosplit directory.

## NAME

ExtUtils::Installed − Inventory management of installed modules

## SYNOPSIS

```
use ExtUtils::Installed;
my ($inst) = ExtUtils::Installed->new();
my (@modules) = $inst->modules();
my (@missing) = $inst->validate("DBI");
my $all_files = $inst->files("DBI");
my $files_below_usr_local = $inst->files("DBI", "all", "/usr/local");
my $all_dirs = $inst->directories("DBI");
my $dirs_below_usr_local = $inst->directory_tree("DBI", "prog");
my $packlist = $inst->packlist("DBI");
```

## DESCRIPTION

ExtUtils::Installed provides a standard way to find out what core and module files have been installed. It uses the information stored in .packlist files created during installation to provide this information. In addition it provides facilities to classify the installed files and to extract directory information from the .packlist files.

## USAGE

The *new()* function searches for all the installed .packlists on the system, and stores their contents. The .packlists can be queried with the functions described below.

## FUNCTIONS

new()
: This takes no parameters, and searches for all the installed .packlists on the system. The packlists are read using the ExtUtils::packlist module.

modules()
: This returns a list of the names of all the installed modules. The perl 'core' is given the special name 'Perl'.

files()
: This takes one mandatory parameter, the name of a module. It returns a list of all the filenames from the package. To obtain a list of core perl files, use the module name 'Perl'. Additional parameters are allowed. The first is one of the strings ''prog'', ''man'' or ''all'', to select either just program files, just manual files or all files. The remaining parameters are a list of directories. The filenames returned will be restricted to those under the specified directories.

directories()
: This takes one mandatory parameter, the name of a module. It returns a list of all the directories from the package. Additional parameters are allowed. The first is one of the strings ''prog'', ''man'' or ''all'', to select either just program directories, just manual directories or all directories. The remaining parameters are a list of directories. The directories returned will be restricted to those under the specified directories. This method returns only the leaf directories that contain files from the specified module.

directory_tree()
: This is identical in operation to *directory()*, except that it includes all the intermediate directories back up to the specified directories.

validate()
: This takes one mandatory parameter, the name of a module. It checks that all the files listed in the modules .packlist actually exist, and returns a list of any missing files. If an optional second argument which evaluates to true is given any missing files will be removed from the .packlist

packlist()
: This returns the ExtUtils::Packlist object for the specified module.

version()
: This returns the version number for the specified module.

## EXAMPLE

See the example in the *ExtUtils::Packlist* manpage.

## AUTHOR

Alan Burlison <Alan.Burlison@uk.sun.com>

**NAME**

ExtUtils::Liblist − determine libraries to use and how to use them

**SYNOPSIS**

```
require ExtUtils::Liblist;

ExtUtils::Liblist::ext($self, $potential_libs, $verbose);
```

**DESCRIPTION**

This utility takes a list of libraries in the form `−llib1 −llib2 −llib3` and prints out lines suitable for inclusion in an extension Makefile. Extra library paths may be included with the form `−L/another/path` this will affect the searches for all subsequent libraries.

It returns an array of four scalar values: EXTRALIBS, BSLOADLIBS, LDLOADLIBS, and LD_RUN_PATH. Some of these don't mean anything on VMS and Win32. See the details about those platform specifics below.

Dependent libraries can be linked in one of three ways:

• For static extensions
  by the ld command when the perl binary is linked with the extension library. See EXTRALIBS below.

• For dynamic extensions
  by the ld command when the shared object is built/linked. See LDLOADLIBS below.

• For dynamic extensions
  by the DynaLoader when the shared object is loaded. See BSLOADLIBS below.

**EXTRALIBS**

List of libraries that need to be linked with when linking a perl binary which includes this extension Only those libraries that actually exist are included. These are written to a file and used when linking perl.

**LDLOADLIBS and LD_RUN_PATH**

List of those libraries which can or must be linked into the shared library when created using ld. These may be static or dynamic libraries. LD_RUN_PATH is a colon separated list of the directories in LDLOADLIBS. It is passed as an environment variable to the process that links the shared library.

**BSLOADLIBS**

List of those libraries that are needed but can be linked in dynamically at run time on this platform. SunOS/Solaris does not need this because ld records the information (from LDLOADLIBS) into the object file. This list is used to create a .bs (bootstrap) file.

**PORTABILITY**

This module deals with a lot of system dependencies and has quite a few architecture specific **if**s in the code.

**VMS implementation**

The version of *ext()* which is executed under VMS differs from the Unix-OS/2 version in several respects:

- Input library and path specifications are accepted with or without the `−l` and `−L` prefices used by Unix linkers. If neither prefix is present, a token is considered a directory to search if it is in fact a directory, and a library to search for otherwise. Authors who wish their extensions to be portable to Unix or OS/2 should use the Unix prefixes, since the Unix-OS/2 version of *ext()* requires them.
- Wherever possible, shareable images are preferred to object libraries, and object libraries to plain object files. In accordance with VMS naming conventions, *ext()* looks for files named *lib*shr and *lib*rtl; it also looks for *lib*lib and lib*lib* to accomodate Unix conventions used in some ported software.

- For each library that is found, an appropriate directive for a linker options file is generated. The return values are space-separated strings of these directives, rather than elements used on the linker command line.
- LDLOADLIBS contains both the libraries found based on `$potential_libs` and the CRTLs, if any, specified in Config.pm. EXTRALIBS contains just those libraries found based on `$potential_libs`. BSLOADLIBS and LD_RUN_PATH are always empty.

In addition, an attempt is made to recognize several common Unix library names, and filter them out or convert them to their VMS equivalents, as appropriate.

In general, the VMS version of *ext()* should properly handle input from extensions originally designed for a Unix or VMS environment. If you encounter problems, or discover cases where the search could be improved, please let us know.

### Win32 implementation

The version of *ext()* which is executed under Win32 differs from the Unix-OS/2 version in several respects:

- Input library and path specifications are accepted with or without the `-l` and `-L` prefices used by Unix linkers. `-lfoo` specifies the library `libfoo.lib` (unless `foo` already starts with `lib`), and `-Ls:ome\dir` specifies a directory to look for the libraries that follow. If neither prefix is present, a token is considered a directory to search if it is in fact a directory, and a library to search for otherwise. The `$Config{lib_ext}` suffix will be appended to any entries that are not directories and don't already have the suffix. Authors who wish their extensions to be portable to Unix or OS/2 should use the Unix prefixes, since the Unix-OS/2 version of *ext()* requires them.
- Entries cannot be plain object files, as many Win32 compilers will not handle object files in the place of libraries.
- If `$potential_libs` is empty, the return value will be empty. Otherwise, the libraries specified by `$Config{libs}` (see Config.pm) will be appended to the list of `$potential_libs`. The libraries will be searched for in the directories specified in `$potential_libs` as well as in `$Config{libpth}`. For each library that is found, a space-separated list of fully qualified library pathnames is generated. You may specify an entry that matches `/:nodefault/i` in `$potential_libs` to disable the appending of default libraries found in `$Config{libs}` (this should be only needed very rarely).
- The libraries specified may be a mixture of static libraries and import libraries (to link with DLLs). Since both kinds are used pretty transparently on the win32 platform, we do not attempt to distinguish between them.
- LDLOADLIBS and EXTRALIBS are always identical under Win32, and BSLOADLIBS and LD_RUN_PATH are always empty (this may change in future).
- You must make sure that any paths and path components are properly surrounded with double-quotes if they contain spaces. For example, `$potential_libs` could be (literally):

  ```
  "-Lc:\Program Files\vc\lib" msvcrt.lib "la test\foo bar.lib"
  ```

  Note how the first and last entries are protected by quotes in order to protect the spaces.

## SEE ALSO

the *ExtUtils::MakeMaker* manpage

**NAME**

ExtUtils::MakeMaker − create an extension Makefile

**SYNOPSIS**

```
use ExtUtils::MakeMaker;
```

```
WriteMakefile( ATTRIBUTE => VALUE [, ...] );
```

which is really

```
MM−>new(\%att)−>flush;
```

**DESCRIPTION**

This utility is designed to write a Makefile for an extension module from a Makefile.PL. It is based on the Makefile.SH model provided by Andy Dougherty and the perl5-porters.

It splits the task of generating the Makefile into several subroutines that can be individually overridden. Each subroutine returns the text it wishes to have written to the Makefile.

MakeMaker is object oriented. Each directory below the current directory that contains a Makefile.PL. Is treated as a separate object. This makes it possible to write an unlimited number of Makefiles with a single invocation of *WriteMakefile()*.

### How To Write A Makefile.PL

The short answer is: Don't.

```
        Always begin with h2xs.
        Always begin with h2xs!
        ALWAYS BEGIN WITH H2XS!
```

even if you're not building around a header file, and even if you don't have an XS component.

Run *h2xs*(1) before you start thinking about writing a module. For so called pm-only modules that consist of `*.pm` files only, h2xs has the `−X` switch. This will generate dummy files of all kinds that are useful for the module developer.

The medium answer is:

```
    use ExtUtils::MakeMaker;
    WriteMakefile( NAME => "Foo::Bar" );
```

The long answer is the rest of the manpage :−)

### Default Makefile Behaviour

The generated Makefile enables the user of the extension to invoke

```
  perl Makefile.PL # optionally "perl Makefile.PL verbose"
  make
  make test        # optionally set TEST_VERBOSE=1
  make install     # See below
```

The Makefile to be produced may be altered by adding arguments of the form `KEY=VALUE`. E.g.

```
  perl Makefile.PL PREFIX=/tmp/myperl5
```

Other interesting targets in the generated Makefile are

```
make config     # to check if the Makefile is up-to-date
make clean      # delete local temp files (Makefile gets renamed)
make realclean  # delete derived files (including ./blib)
make ci         # check in all the files in the MANIFEST file
make dist       # see below the Distribution Support section
```

**make test**

MakeMaker checks for the existence of a file named *test.pl* in the current directory and if it exists it adds commands to the test target of the generated Makefile that will execute the script with the proper set of perl `-I` options.

MakeMaker also checks for any files matching *glob* ("t/*.t"). It will add commands to the test target of the generated Makefile that execute all matching files via the the *Test::Harness* manpage module with the `-I` switches set correctly.

**make testdb**

A useful variation of the above is the target `testdb`. It runs the test under the Perl debugger (see the *perldebug* manpage). If the file *test.pl* exists in the current directory, it is used for the test.

If you want to debug some other testfile, set `TEST_FILE` variable thusly:

```
make testdb TEST_FILE=t/mytest.t
```

By default the debugger is called using `-d` option to perl. If you want to specify some other option, set `TESTDB_SW` variable:

```
make testdb TESTDB_SW=-Dx
```

**make install**

make alone puts all relevant files into directories that are named by the macros INST_LIB, INST_ARCHLIB, INST_SCRIPT, INST_MAN1DIR, and INST_MAN3DIR. All these default to something below ./blib if you are *not* building below the perl source directory. If you *are* building below the perl source, INST_LIB and INST_ARCHLIB default to
../../lib, and INST_SCRIPT is not defined.

The *install* target of the generated Makefile copies the files found below each of the INST_* directories to their INSTALL* counterparts. Which counterparts are chosen depends on the setting of INSTALLDIRS according to the following table:

```
                         INSTALLDIRS set to
                      perl              site

  INST_ARCHLIB     INSTALLARCHLIB    INSTALLSITEARCH
  INST_LIB         INSTALLPRIVLIB    INSTALLSITELIB
  INST_BIN                   INSTALLBIN
  INST_SCRIPT               INSTALLSCRIPT
  INST_MAN1DIR              INSTALLMAN1DIR
  INST_MAN3DIR              INSTALLMAN3DIR
```

The INSTALL... macros in turn default to their `%Config` ($Config{installprivlib}, `$Config`{installarchlib}, etc.) counterparts.

You can check the values of these variables on your system with

```
perl '-V:install.*'
```

And to check the sequence in which the library directories are searched by perl, run

```
perl -le 'print join $/, @INC'
```

### PREFIX and LIB attribute

PREFIX and LIB can be used to set several INSTALL* attributes in one go. The quickest way to install a module in a non-standard place might be

```
perl Makefile.PL LIB=~/lib
```

This will install the module's architecture-independent files into ~/lib, the architecture-dependent files into ~/lib/$archname/auto.

Another way to specify many INSTALL directories with a single parameter is PREFIX.

```
perl Makefile.PL PREFIX=~
```

This will replace the string specified by `$Config`{prefix} in all `$Config`{install*} values.

Note, that in both cases the tilde expansion is done by MakeMaker, not by perl by default, nor by make. Conflicts between parmeters LIB, PREFIX and the various INSTALL* arguments are resolved so that XXX

If the user has superuser privileges, and is not working on AFS (Andrew File System) or relatives, then the defaults for INSTALLPRIVLIB, INSTALLARCHLIB, INSTALLSCRIPT, etc. will be appropriate, and this incantation will be the best:

```
perl Makefile.PL; make; make test
make install
```

make install per default writes some documentation of what has been done into the file `$(INSTALLARCHLIB)/perllocal.pod`. This feature can be bypassed by calling make pure_install.

### AFS users

will have to specify the installation directories as these most probably have changed since perl itself has been installed. They will have to do this by calling

```
perl Makefile.PL INSTALLSITELIB=/afs/here/today \
    INSTALLSCRIPT=/afs/there/now INSTALLMAN3DIR=/afs/for/manpages
make
```

Be careful to repeat this procedure every time you recompile an extension, unless you are sure the AFS installation directories are still valid.

### Static Linking of a new Perl Binary

An extension that is built with the above steps is ready to use on systems supporting dynamic loading. On systems that do not support dynamic loading, any newly created extension has to be linked together with the available resources. MakeMaker supports the linking process by creating appropriate targets in the Makefile whenever an extension is built. You can invoke the corresponding section of the makefile with

```
make perl
```

That produces a new perl binary in the current directory with all extensions linked in that can be found in INST_ARCHLIB , SITELIBEXP, and PERL_ARCHLIB. To do that, MakeMaker writes a new Makefile, on UNIX, this is called Makefile.aperl (may be system dependent). If you want to force the creation of a new perl, it is recommended, that you delete this Makefile.aperl, so the directories are searched-through for linkable libraries again.

The binary can be installed into the directory where perl normally resides on your machine with

```
make inst_perl
```

To produce a perl binary with a different name than `perl`, either say

```
perl Makefile.PL MAP_TARGET=myperl
make myperl
make inst_perl
```

or say

```
perl Makefile.PL
make myperl MAP_TARGET=myperl
make inst_perl MAP_TARGET=myperl
```

In any case you will be prompted with the correct invocation of the `inst_perl` target that installs the new binary into INSTALLBIN.

make inst_perl per default writes some documentation of what has been done into the file `$(INSTALLARCHLIB)/perllocal.pod`. This can be bypassed by calling make pure_inst_perl.

Warning: the inst_perl: target will most probably overwrite your existing perl binary. Use with care!

Sometimes you might want to build a statically linked perl although your system supports dynamic loading. In this case you may explicitly set the linktype with the invocation of the Makefile.PL or make:

```
perl Makefile.PL LINKTYPE=static    # recommended
```

or

```
make LINKTYPE=static                # works on most systems
```

**Determination of Perl Library and Installation Locations**

MakeMaker needs to know, or to guess, where certain things are located. Especially INST_LIB and INST_ARCHLIB (where to put the files during the *make*(1) run), PERL_LIB and PERL_ARCHLIB (where to read existing modules from), and PERL_INC (header files and `libperl*.*`).

Extensions may be built either using the contents of the perl source directory tree or from the installed perl library. The recommended way is to build extensions after you have run 'make install' on perl itself. You can do that in any directory on your hard disk that is not below the perl source tree. The support for extensions below the ext directory of the perl distribution is only good for the standard extensions that come with perl.

If an extension is being built below the `ext/` directory of the perl source then MakeMaker will set PERL_SRC automatically (e.g., `../..`). If PERL_SRC is defined and the extension is recognized as a standard extension, then other variables default to the following:

```
PERL_INC     = PERL_SRC
PERL_LIB     = PERL_SRC/lib
PERL_ARCHLIB = PERL_SRC/lib
INST_LIB     = PERL_LIB
INST_ARCHLIB = PERL_ARCHLIB
```

If an extension is being built away from the perl source then MakeMaker will leave PERL_SRC undefined and default to using the installed copy of the perl library. The other variables default to the following:

```
PERL_INC     = $archlibexp/CORE
PERL_LIB     = $privlibexp
PERL_ARCHLIB = $archlibexp
INST_LIB     = ./blib/lib
INST_ARCHLIB = ./blib/arch
```

If perl has not yet been installed then PERL_SRC can be defined on the command line as shown in the previous section.

### Which architecture dependent directory?

If you don't want to keep the defaults for the INSTALL* macros, MakeMaker helps you to minimize the typing needed: the usual relationship between INSTALLPRIVLIB and INSTALLARCHLIB is determined by Configure at perl compilation time. MakeMaker supports the user who sets INSTALLPRIVLIB. If INSTALLPRIVLIB is set, but INSTALLARCHLIB not, then MakeMaker defaults the latter to be the same subdirectory of INSTALLPRIVLIB as Configure decided for the counterparts in `%Config` , otherwise it defaults to INSTALLPRIVLIB. The same relationship holds for INSTALLSITELIB and INSTALLSITEARCH.

MakeMaker gives you much more freedom than needed to configure internal variables and get different results. It is worth to mention, that *make* (1) also lets you configure most of the variables that are used in the Makefile. But in the majority of situations this will not be necessary, and should only be done, if the author of a package recommends it (or you know what you're doing).

### Using Attributes and Parameters

The following attributes can be specified as arguments to *WriteMakefile()* or as NAME=VALUE pairs on the command line:

C
: Ref to array of *.c file names. Initialised from a directory scan and the values portion of the XS attribute hash. This is not currently used by MakeMaker but may be handy in Makefile.PLs.

CCFLAGS
: String that will be included in the compiler call command line between the arguments INC and OPTIMIZE.

CONFIG
: Arrayref. E.g. [*qw* (archname manext)] defines ARCHNAME & MANEXT from config.sh. MakeMaker will add to CONFIG the following values anyway: ar cc cccdlflags ccdlflags dlext dlsrc ld lddlflags ldflags libc lib_ext obj_ext ranlib sitelibexp sitearchexp so

CONFIGURE
: CODE reference. The subroutine should return a hash reference. The hash may contain further attributes, e.g. {LIBS => ...}, that have to be determined by some evaluation method.

DEFINE
: Something like `"-DHAVE_UNISTD_H"`

DIR
: Ref to array of subdirectories containing Makefile.PLs e.g. [ 'sdbm' ] in ext/SDBM_File

DISTNAME
: Your name for distributing the package (by tar file). This defaults to NAME above.

DL_FUNCS
: Hashref of symbol names for routines to be made available as universal symbols. Each key/value pair consists of the package name and an array of routine names in that package. Used only under AIX (export lists) and VMS (linker options) at present. The routine names supplied will be expanded in the same way as XSUB names are expanded by the *XS()* macro. Defaults to

```
{"$(NAME)" => ["boot_$(NAME)" ] }
```

e.g.

```
{"RPC" => [qw( boot_rpcb rpcb_gettime getnetconfigent )],
 "NetconfigPtr" => [ 'DESTROY'] }
```

DL_VARS
Array of symbol names for variables to be made available as universal symbols. Used only under AIX (export lists) and VMS (linker options) at present. Defaults to []. (e.g. [ *qw*( Foo_version Foo_numstreams Foo_tree ) ])

EXCLUDE_EXT
Array of extension names to exclude when doing a static build. This is ignored if INCLUDE_EXT is present. Consult INCLUDE_EXT for more details. (e.g. [ *qw* ( Socket POSIX ) ] )

This attribute may be most useful when specified as a string on the commandline: perl Makefile.PL EXCLUDE_EXT='Socket Safe'

EXE_FILES
Ref to array of executable files. The files will be copied to the INST_SCRIPT directory. Make realclean will delete them from there again.

NO_VC
In general any generated Makefile checks for the current version of MakeMaker and the version the Makefile was built under. If NO_VC is set, the version check is neglected. Do not write this into your Makefile.PL, use it interactively instead.

FIRST_MAKEFILE
The name of the Makefile to be produced. Defaults to the contents of MAKEFILE, but can be overridden. This is used for the second Makefile that will be produced for the MAP_TARGET.

FULLPERL
Perl binary able to run this extension.

H
Ref to array of *.h file names. Similar to C.

IMPORTS
IMPORTS is only used on OS/2.

INC
Include file dirs eg: `"-I/usr/5include -I/path/to/inc"`

INCLUDE_EXT
Array of extension names to be included when doing a static build. MakeMaker will normally build with all of the installed extensions when doing a static build, and that is usually the desired behavior. If INCLUDE_EXT is present then MakeMaker will build only with those extensions which are explicitly mentioned. (e.g. [ *qw* ( Socket POSIX ) ])

It is not necessary to mention DynaLoader or the current extension when filling in INCLUDE_EXT. If the INCLUDE_EXT is mentioned but is empty then only DynaLoader and the current extension will be included in the build.

This attribute may be most useful when specified as a string on the commandline: perl Makefile.PL INCLUDE_EXT='POSIX Socket Devel::Peek'

INSTALLARCHLIB
Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to perl.

INSTALLBIN
Directory to install binary files (e.g. tkperl) into.

INSTALLDIRS
Determines which of the two sets of installation directories to choose: installprivlib and installarchlib versus installsitelib and installsitearch. The first pair is chosen with INSTALLDIRS=perl, the second with INSTALLDIRS=site. Default is site.

INSTALLMAN1DIR
This directory gets the man pages at 'make install' time. Defaults to `$Config`{installman1dir}.

INSTALLMAN3DIR
This directory gets the man pages at 'make install' time. Defaults to `$Config`{installman3dir}.

INSTALLPRIVLIB
Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to perl.

INSTALLSCRIPT
Used by 'make install' which copies files from INST_SCRIPT to this directory.

INSTALLSITELIB
Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to site (default).

INSTALLSITEARCH
Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to site (default).

INST_ARCHLIB
Same as INST_LIB for architecture dependent files.

INST_BIN
Directory to put real binary files during 'make'. These will be copied to INSTALLBIN during 'make install'

INST_EXE
Old name for INST_SCRIPT. Deprecated. Please use INST_SCRIPT if you need to use it.

INST_LIB
Directory where we put library files of this extension while building it.

INST_MAN1DIR
Directory to hold the man pages at 'make' time

INST_MAN3DIR
Directory to hold the man pages at 'make' time

INST_SCRIPT
Directory, where executable files should be installed during 'make'. Defaults to "./blib/bin", just to have a dummy location during testing. make install will copy the files in INST_SCRIPT to INSTALLSCRIPT.

LDFROM
defaults to "$(OBJECT)" and is used in the ld command to specify what files to link/load from (also see dynamic_lib below for how to specify ld flags)

LIBPERL_A
The filename of the perllibrary that will be used together with this extension. Defaults to libperl.a.

LIB
LIB can only be set at `perl Makefile.PL` time. It has the effect of setting both INSTALLPRIVLIB and INSTALLSITELIB to that value regardless any

LIBS
An anonymous array of alternative library specifications to be searched for (in order) until at least one library is found. E.g.

```
'LIBS' => ["-lgdbm", "-ldbm -lfoo", "-L/path -ldbm.nfs"]
```

Mind, that any element of the array contains a complete set of arguments for the ld command. So do not specify

```
'LIBS' => ["-ltcl", "-ltk", "-lX11"]
```

See ODBM_File/Makefile.PL for an example, where an array is needed. If you specify a scalar as in

```
'LIBS' => "-ltcl -ltk -lX11"
```

MakeMaker will turn it into an array with one element.

LINKTYPE
'static' or 'dynamic' (default unless usedl=undef in config.sh). Should only be used to force static linking (also see linkext below).

MAKEAPERL
Boolean which tells MakeMaker, that it should include the rules to make a perl. This is handled automatically as a switch by MakeMaker. The user normally does not need it.

MAKEFILE
The name of the Makefile to be produced.

MAN1PODS
Hashref of pod-containing files. MakeMaker will default this to all EXE_FILES files that include POD directives. The files listed here will be converted to man pages and installed as was requested at Configure time.

MAN3PODS
Hashref of .pm and .pod files. MakeMaker will default this to all
.pod and any .pm files that include POD directives. The files listed here will be converted to man pages and installed as was requested at Configure time.

MAP_TARGET
If it is intended, that a new perl binary be produced, this variable may hold a name for that binary. Defaults to perl

MYEXTLIB
If the extension links to a library that it builds set this to the name of the library (see SDBM_File)

NAME
Perl module name for this extension (DBD::Oracle). This will default to the directory name but should be explicitly defined in the Makefile.PL.

NEEDS_LINKING
MakeMaker will figure out, if an extension contains linkable code anywhere down the directory tree, and will set this variable accordingly, but you can speed it up a very little bit, if you define this boolean variable yourself.

NOECHO
Defaults to `@`. By setting it to an empty string you can generate a Makefile that echos all commands. Mainly used in debugging MakeMaker itself.

NORECURS
Boolean. Attribute to inhibit descending into subdirectories.

OBJECT
List of object files, defaults to '$(BASEEXT)$(OBJ_EXT)', but can be a long string containing all object files, e.g. "tkpBind.o tkpButton.o tkpCanvas.o"

OPTIMIZE
Defaults to `-O`. Set it to `-g` to turn debugging on. The flag is passed to subdirectory makes.

PERL
Perl binary for tasks that can be done by miniperl

PERLMAINCC
The call to the program that is able to compile perlmain.c. Defaults to $(CC).

PERL_ARCHLIB
Same as above for architecture dependent files

PERL_LIB
Directory containing the Perl library to use.

PERL_SRC
Directory containing the Perl source code (use of this should be avoided, it may be undefined)

PERM_RW
Desired Permission for read/writable files. Defaults to `644`. See also the `perm_rw` entry in the *MM_Unix* manpage.

PERM_RWX
Desired permission for executable files. Defaults to `755`. See also the `perm_rwx` entry in the *MM_Unix* manpage.

PL_FILES
Ref to hash of files to be processed as perl programs. MakeMaker will default to any found *.PL file (except Makefile.PL) being keys and the basename of the file being the value. E.g.

```
{'foobar.PL' => 'foobar'}
```

The *.PL files are expected to produce output to the target files themselves.

PM
Hashref of .pm files and *.pl files to be installed. e.g.

```
{'name_of_file.pm' => '$(INST_LIBDIR)/install_as.pm'}
```

By default this will include *.pm and *.pl and the files found in the PMLIBDIRS directories. Defining PM in the Makefile.PL will override PMLIBDIRS.

PMLIBDIRS
Ref to array of subdirectories containing library files. Defaults to [ 'lib', $(BASEEXT) ]. The directories will be scanned and *any* files they contain will be installed in the corresponding location in the library. A *libscan()* method can be used to alter the behaviour. Defining PM in the Makefile.PL will override PMLIBDIRS.

PREFIX
Can be used to set the three INSTALL* attributes in one go (except for probably INSTALLMAN1DIR, if it is not below PREFIX according to `%Config`). They will have PREFIX as a common directory node and will branch from that node into lib/, lib/ARCHNAME or whatever Configure decided at the build time of your perl (unless you override one of them, of course).

PREREQ_PM
Hashref: Names of modules that need to be available to run this extension (e.g. Fcntl for SDBM_File) are the keys of the hash and the desired version is the value. If the required version number is 0, we only check if any version is installed already.

SKIP
Arryref. E.g. [*qw* (name1 name2)] skip (do not write) sections of the Makefile. Caution! Do not use the SKIP attribute for the neglectible speedup. It may seriously damage the resulting Makefile. Only use it, if you really need it.

TYPEMAPS

Ref to array of typemap file names. Use this when the typemaps are in some directory other than the current directory or when they are not named **typemap**. The last typemap in the list takes precedence. A typemap in the current directory has highest precedence, even if it isn't listed in TYPEMAPS. The default system typemap has lowest precedence.

VERSION

Your version number for distributing the package. This defaults to 0.1.

VERSION_FROM

Instead of specifying the VERSION in the Makefile.PL you can let MakeMaker parse a file to determine the version number. The parsing routine requires that the file named by VERSION_FROM contains one single line to compute the version number. The first line in the file that contains the regular expression

```
/([\$*])(([\w\:\']*)\bVERSION)\b.*\=/
```

will be evaluated with *eval()* and the value of the named variable **after** the *eval()* will be assigned to the VERSION attribute of the MakeMaker object. The following lines will be parsed o.k.:

```
$VERSION = '1.00';
*VERSION = \'1.01';
( $VERSION ) = '$Revision: 1.222 $ ' =~ /\$Revision:\s+([^\s]+)/;
$FOO::VERSION = '1.10';
*FOO::VERSION = \'1.11';
```

but these will fail:

```
my $VERSION = '1.01';
local $VERSION = '1.02';
local $FOO::VERSION = '1.30';
```

The file named in VERSION_FROM is not added as a dependency to Makefile. This is not really correct, but it would be a major pain during development to have to rewrite the Makefile for any smallish change in that file. If you want to make sure that the Makefile contains the correct VERSION macro after any change of the file, you would have to do something like

```
depend => { Makefile => '$(VERSION_FROM)' }
```

See attribute `depend` below.

XS

Hashref of .xs files. MakeMaker will default this. e.g.

```
{'name_of_file.xs' => 'name_of_file.c'}
```

The .c files will automatically be included in the list of files deleted by a make clean.

XSOPT

String of options to pass to xsubpp. This might include `-C++` or `-extern`. Do not include typemaps here; the TYPEMAP parameter exists for that purpose.

XSPROTOARG

May be set to an empty string, which is identical to `-prototypes`, or `-noprototypes`. See the xsubpp documentation for details. MakeMaker defaults to the empty string.

XS_VERSION

Your version number for the .xs file of this package. This defaults to the value of the VERSION attribute.

### Additional lowercase attributes

can be used to pass parameters to the methods which implement that part of the Makefile.

clean

```
{FILES => "*.xyz foo"}
```

depend

```
{ANY_TARGET => ANY_DEPENDECY, ...}
```

dist

```
{TARFLAGS => 'cvfF', COMPRESS => 'gzip', SUFFIX => '.gz',
SHAR => 'shar -m', DIST_CP => 'ln', ZIP => '/bin/zip',
ZIPFLAGS => '-rl', DIST_DEFAULT => 'private tardist' }
```

If you specify COMPRESS, then SUFFIX should also be altered, as it is needed to tell make the target file of the compression. Setting DIST_CP to ln can be useful, if you need to preserve the timestamps on your files. DIST_CP can take the values 'cp', which copies the file, 'ln', which links the file, and 'best' which copies symbolic links and links the rest. Default is 'best'.

dynamic_lib

```
{ARMAYBE => 'ar', OTHERLDFLAGS => '...', INST_DYNAMIC_DEP => '...'}
```

installpm

Deprecated as of MakeMaker 5.23. See the `pm_to_blib` entry in the *ExtUtils::MM_Unix* manpage.

linkext

```
{LINKTYPE => 'static', 'dynamic' or ''}
```

NB: Extensions that have nothing but *.pm files had to say

```
{LINKTYPE => ''}
```

with Pre-5.0 MakeMakers. Since version 5.00 of MakeMaker such a line can be deleted safely. MakeMaker recognizes, when there's nothing to be linked.

macro

```
{ANY_MACRO => ANY_VALUE, ...}
```

realclean

```
{FILES => '$(INST_ARCHAUTODIR)/*.xyz'}
```

tool_autosplit

```
{MAXLEN =E<gt> 8}
```

### Overriding MakeMaker Methods

If you cannot achieve the desired Makefile behaviour by specifying attributes you may define private subroutines in the Makefile.PL. Each subroutines returns the text it wishes to have written to the Makefile. To override a section of the Makefile you can either say:

```
        sub MY::c_o { "new literal text" }
```

or you can edit the default by saying something like:

```
        sub MY::c_o {
            package MY; # so that "SUPER" works right
            my $inherited = shift->SUPER::c_o(@_);
            $inherited =~ s/old text/new text/;
            $inherited;
        }
```

If you are running experiments with embedding perl as a library into other applications, you might find MakeMaker is not sufficient. You'd better have a look at ExtUtils::Embed which is a collection of utilities for embedding.

If you still need a different solution, try to develop another subroutine that fits your needs and submit the diffs to *perl5-porters@perl.org* or *comp.lang.perl.moderated* as appropriate.

For a complete description of all MakeMaker methods see the *ExtUtils::MM_Unix* manpage.

Here is a simple example of how to add a new target to the generated Makefile:

```
    sub MY::postamble {
        '
    $(MYEXTLIB): sdbm/Makefile
            cd sdbm && $(MAKE) all
    ';
    }
```

**Hintsfile support**

MakeMaker.pm uses the architecture specific information from Config.pm. In addition it evaluates architecture specific hints files in a `hints/` directory. The hints files are expected to be named like their counterparts in `PERL_SRC/hints`, but with an `.pl` file name extension (eg. `next_3_2.pl`). They are simply `eval`ed by MakeMaker within the *WriteMakefile()* subroutine, and can be used to execute commands as well as to include special variables. The rules which hintsfile is chosen are the same as in Configure.

The hintsfile is *eval()*ed immediately after the arguments given to WriteMakefile are stuffed into a hash reference `$self` but before this reference becomes blessed. So if you want to do the equivalent to override or create an attribute you would say something like

```
    $self->{LIBS} = ['-ldbm -lucb -lc'];
```

**Distribution Support**

For authors of extensions MakeMaker provides several Makefile targets. Most of the support comes from the ExtUtils::Manifest module, where additional documentation can be found.

make distcheck
: reports which files are below the build directory but not in the MANIFEST file and vice versa. (See *ExtUtils::Manifest::fullcheck()* for details)

make skipcheck
: reports which files are skipped due to the entries in the `MANIFEST.SKIP` file (See *ExtUtils::Manifest::skipcheck()* for details)

make distclean
: does a realclean first and then the distcheck. Note that this is not needed to build a new distribution as long as you are sure, that the MANIFEST file is ok.

make manifest
: rewrites the MANIFEST file, adding all remaining files found (See *ExtUtils::Manifest::mkmanifest()* for details)

make distdir
: Copies all the files that are in the MANIFEST file to a newly created directory with the name `$(DISTNAME)-$(VERSION)`. If that directory exists, it will be removed first.

make disttest
: Makes a distdir first, and runs a `perl Makefile.PL`, a make, and a make test in that directory.

make tardist
: First does a distdir. Then a command $(PREOP) which defaults to a null command, followed by $(TOUNIX), which defaults to a null command under UNIX, and will convert files in distribution directory to UNIX format otherwise. Next it runs `tar` on that directory into a tarfile and deletes the directory. Finishes with a command $(POSTOP) which defaults to a null command.

make dist
: Defaults to $(DIST_DEFAULT) which in turn defaults to tardist.

make uutardist
: Runs a tardist first and uuencodes the tarfile.

make shdist
: First does a distdir. Then a command $(PREOP) which defaults to a null command. Next it runs `shar` on that directory into a sharfile and deletes the intermediate directory again. Finishes with a command $(POSTOP) which defaults to a null command. Note: For shdist to work properly a `shar` program that can handle directories is mandatory.

make zipdist
: First does a distdir. Then a command $(PREOP) which defaults to a null command. Runs `$(ZIP) $(ZIPFLAGS)` on that directory into a zipfile. Then deletes that directory. Finishes with a command $(POSTOP) which defaults to a null command.

make ci
: Does a $(CI) and a $(RCS_LABEL) on all files in the MANIFEST file.

Customization of the dist targets can be done by specifying a hash reference to the dist attribute of the WriteMakefile call. The following parameters are recognized:

```
CI           ('ci -u')
COMPRESS     ('gzip --best')
POSTOP       ('@ :')
PREOP        ('@ :')
TO_UNIX      (depends on the system)
RCS_LABEL    ('rcs -q -Nv$(VERSION_SYM):')
SHAR         ('shar')
SUFFIX       ('.gz')
TAR          ('tar')
TARFLAGS     ('cvf')
ZIP          ('zip')
ZIPFLAGS     ('-r')
```

An example:

```
WriteMakefile( 'dist' => { COMPRESS=>"bzip2", SUFFIX=>".bz2" })
```

### Disabling an extension

If some events detected in *Makefile.PL* imply that there is no way to create the Module, but this is a normal state of things, then you can create a *Makefile* which does nothing, but succeeds on all the "usual" build targets. To do so, use

```
ExtUtils::MakeMaker::WriteEmptyMakefile();
```

instead of *WriteMakefile()*.

This may be useful if other modules expect this module to be *built* OK, as opposed to *work* OK (say, this system-dependent module builds in a subdirectory of some other distribution, or is listed as a dependency in a CPAN::Bundle, but the functionality is supported by different means on the current architecture).

## SEE ALSO

ExtUtils::MM_Unix, ExtUtils::Manifest, ExtUtils::testlib, ExtUtils::Install, ExtUtils::Embed

## AUTHORS

Andy Dougherty <*doughera@lafcol.lafayette.edu*>, Andreas König <*A.Koenig@franz.ww.TU−Berlin.DE*>, Tim Bunce <*Tim.Bunce@ig.co.uk*>. VMS support by Charles Bailey <*bailey@genetics.upenn.edu*>. OS/2 support by Ilya Zakharevich <*ilya@math.ohio-state.edu*>. Contact the makemaker mailing list `mailto:makemaker@franz.ww.tu-berlin.de`, if you have any questions.

**NAME**

ExtUtils::Manifest – utilities to write and check a MANIFEST file

**SYNOPSIS**

```
require ExtUtils::Manifest;

ExtUtils::Manifest::mkmanifest;

ExtUtils::Manifest::manicheck;

ExtUtils::Manifest::filecheck;

ExtUtils::Manifest::fullcheck;

ExtUtils::Manifest::skipcheck;

ExtUtild::Manifest::manifind();

ExtUtils::Manifest::maniread($file);

ExtUtils::Manifest::manicopy($read,$target,$how);
```

**DESCRIPTION**

*Mkmanifest()* writes all files in and below the current directory to a file named in the global variable `$ExtUtils::Manifest::MANIFEST` (which defaults to `MANIFEST`) in the current directory. It works similar to

```
find . -print
```

but in doing so checks each line in an existing `MANIFEST` file and includes any comments that are found in the existing `MANIFEST` file in the new one. Anything between white space and an end of line within a `MANIFEST` file is considered to be a comment. Filenames and comments are seperated by one or more TAB characters in the output. All files that match any regular expression in a file `MANIFEST.SKIP` (if such a file exists) are ignored.

*Manicheck()* checks if all the files within a `MANIFEST` in the current directory really do exist. It only reports discrepancies and exits silently if MANIFEST and the tree below the current directory are in sync.

*Filecheck()* finds files below the current directory that are not mentioned in the `MANIFEST` file. An optional file `MANIFEST.SKIP` will be consulted. Any file matching a regular expression in such a file will not be reported as missing in the `MANIFEST` file.

*Fullcheck()* does both a *manicheck()* and a *filecheck()*.

*Skipcheck()* lists all the files that are skipped due to your `MANIFEST.SKIP` file.

*Manifind()* retruns a hash reference. The keys of the hash are the files found below the current directory.

*Maniread*($file) reads a named `MANIFEST` file (defaults to `MANIFEST` in the current directory) and returns a HASH reference with files being the keys and comments being the values of the HASH. Blank lines and lines which start with `#` in the `MANIFEST` file are discarded.

*Manicopy($read,$target,$how)* copies the files that are the keys in the HASH *%$read* to the named target directory. The HASH reference *$read* is typically returned by the *maniread()* function. This function is useful for producing a directory tree identical to the intended distribution tree. The third parameter `$how` can be used to specify a different methods of ‘‘copying’’. Valid values are `cp`, which actually copies the files, `ln` which creates hard links, and `best` which mostly links the files but copies any symbolic link to make a tree without any symbolic link. Best is the default.

**MANIFEST.SKIP**

The file MANIFEST.SKIP may contain regular expressions of files that should be ignored by *mkmanifest()* and *filecheck()*. The regular expressions should appear one on each line. Blank lines and lines which start with `#` are skipped. Use `\#` if you need a regular expression to start with a sharp character. A typical example:

```
\bRCS\b
^MANIFEST\.
^Makefile$
~$
\.html$
\.old$
^blib/
^MakeMaker-\d
```

**EXPORT_OK**

`&mkmanifest`, `&manicheck`, `&filecheck`, `&fullcheck`, `&maniread`, and `&manicopy` are exportable.

**GLOBAL VARIABLES**

`$ExtUtils::Manifest::MANIFEST` defaults to `MANIFEST`. Changing it results in both a different `MANIFEST` and a different `MANIFEST.SKIP` file. This is useful if you want to maintain different distributions for different audiences (say a user version and a developer version including RCS).

`$ExtUtils::Manifest::Quiet` defaults to 0. If set to a true value, all functions act silently.

**DIAGNOSTICS**

All diagnostic output is sent to `STDERR`.

`Not in MANIFEST:` *file*
: is reported if a file is found, that is missing in the `MANIFEST` file which is excluded by a regular expression in the file `MANIFEST.SKIP`.

`No such file:` *file*
: is reported if a file mentioned in a `MANIFEST` file does not exist.

`MANIFEST:` *$!*
: is reported if `MANIFEST` could not be opened.

`Added to MANIFEST:` *file*
: is reported by *mkmanifest()* if `$Verbose` is set and a file is added to MANIFEST. `$Verbose` is set to 1 by default.

**SEE ALSO**

the *ExtUtils::MakeMaker* manpage which has handy targets for most of the functionality.

**AUTHOR**

Andreas Koenig <*koenig@franz.ww.TU−Berlin.DE*>

## NAME

ExtUtils::Miniperl, writemain – write the C code for perlmain.c

## SYNOPSIS

```
use ExtUtils::Miniperl;

writemain(@directories);
```

## DESCRIPTION

This whole module is written when perl itself is built from a script called minimod.PL. In case you want to patch it, please patch minimod.PL in the perl distribution instead.

*writemain()* takes an argument list of directories containing archive libraries that relate to perl modules and should be linked into a new perl binary. It writes to STDOUT a corresponding perlmain.c file that is a plain C file containing all the bootstrap code to make the modules associated with the libraries available from within perl.

The typical usage is from within a Makefile generated by ExtUtils::MakeMaker. So under normal circumstances you won't have to deal with this module directly.

## SEE ALSO

the *ExtUtils::MakeMaker* manpage

**NAME**

ExtUtils::Mkbootstrap − make a bootstrap file for use by DynaLoader

**SYNOPSIS**

```
mkbootstrap
```

**DESCRIPTION**

Mkbootstrap typically gets called from an extension Makefile.

There is no `*.bs` file supplied with the extension. Instead a `*_BS` file which has code for the special cases, like posix for berkeley db on the NeXT.

This file will get parsed, and produce a maybe empty `@DynaLoader::dl_resolve_using` array for the current architecture. That will be extended by `$BSLOADLIBS`, which was computed by *ExtUtils::Liblist::ext()*. If this array still is empty, we do nothing, else we write a .bs file with an `@DynaLoader::dl_resolve_using` array.

The `*_BS` file can put some code into the generated `*.bs` file by placing it in `$bscode`. This is a handy 'escape' mechanism that may prove useful in complex situations.

If `@DynaLoader::dl_resolve_using` contains `-L*` or `-l*` entries then Mkbootstrap will automatically add a *dl_findfile()* call to the generated `*.bs` file.

## NAME

ExtUtils::Mksymlists − write linker options files for dynamic extension

## SYNOPSIS

```
use ExtUtils::Mksymlists;
Mksymlists({ NAME     => $name ,
             DL_VARS  => [ $var1, $var2, $var3 ],
             DL_FUNCS => { $pkg1 => [ $func1, $func2 ],
                           $pkg2 => [ $func3 ] });
```

## DESCRIPTION

`ExtUtils::Mksymlists` produces files used by the linker under some OSs during the creation of shared libraries for dynamic extensions. It is normally called from a MakeMaker-generated Makefile when the extension is built. The linker option file is generated by calling the function `Mksymlists`, which is exported by default from `ExtUtils::Mksymlists`. It takes one argument, a list of key-value pairs, in which the following keys are recognized:

NAME
: This gives the name of the extension (*e.g.* Tk::Canvas) for which the linker option file will be produced.

DL_FUNCS
: This is identical to the DL_FUNCS attribute available via MakeMaker, from which it is usually taken. Its value is a reference to an associative array, in which each key is the name of a package, and each value is an a reference to an array of function names which should be exported by the extension. For instance, one might say `DL_FUNCS => { Homer::Iliad => [ qw(trojans greeks) ], Homer::Odyssey => [ qw(travellers family suitors) ] }`. The function names should be identical to those in the XSUB code; `Mksymlists` will alter the names written to the linker option file to match the changes made by *xsubpp*. In addition, if none of the functions in a list begin with the string **boot_**, `Mksymlists` will add a bootstrap function for that package, just as xsubpp does. (If a **boot_<pkg>** function is present in the list, it is passed through unchanged.) If DL_FUNCS is not specified, it defaults to the bootstrap function for the extension specified in NAME.

DL_VARS
: This is identical to the DL_VARS attribute available via MakeMaker, and, like DL_FUNCS, it is usually specified via MakeMaker. Its value is a reference to an array of variable names which should be exported by the extension.

FILE
: This key can be used to specify the name of the linker option file (minus the OS−specific extension), if for some reason you do not want to use the default value, which is the last word of the NAME attribute (*e.g.* for Tk::Canvas, FILE defaults to 'Canvas').

FUNCLIST
: This provides an alternate means to specify function names to be exported from the extension. Its value is a reference to an array of function names to be exported by the extension. These names are passed through unaltered to the linker options file.

DLBASE
: This item specifies the name by which the linker knows the extension, which may be different from the name of the extension itself (for instance, some linkers add an '_' to the name of the extension). If it is not specified, it is derived from the NAME attribute. It is presently used only by OS2.

When calling `Mksymlists`, one should always specify the NAME attribute. In most cases, this is all that's necessary. In the case of unusual extensions, however, the other attributes can be used to provide additional information to the linker.

**AUTHOR**

Charles Bailey *<bailey@genetics.upenn.edu>*

**REVISION**

Last revised 14-Feb-1996, for Perl 5.002.

## NAME

ExtUtils::MM_OS2 − methods to override UN*X behaviour in ExtUtils::MakeMaker

## SYNOPSIS

```
use ExtUtils::MM_OS2; # Done internally by ExtUtils::MakeMaker if needed
```

## DESCRIPTION

See ExtUtils::MM_Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

**NAME**

ExtUtils::MM_Unix − methods used by ExtUtils::MakeMaker

**SYNOPSIS**

```
require ExtUtils::MM_Unix;
```

**DESCRIPTION**

The methods provided by this package are designed to be used in conjunction with ExtUtils::MakeMaker. When MakeMaker writes a Makefile, it creates one or more objects that inherit their methods from a package MM. MM itself doesn't provide any methods, but it ISA ExtUtils::MM_Unix class. The inheritance tree of MM lets operating specific packages take the responsibility for all the methods provided by MM_Unix. We are trying to reduce the number of the necessary overrides by defining rather primitive operations within ExtUtils::MM_Unix.

If you are going to write a platform specific MM package, please try to limit the necessary overrides to primitive methods, and if it is not possible to do so, let's work out how to achieve that gain.

If you are overriding any of these methods in your Makefile.PL (in the MY class), please report that to the makemaker mailing list. We are trying to minimize the necessary method overrides and switch to data driven Makefile.PLs wherever possible. In the long run less methods will be overridable via the MY class.

**METHODS**

The following description of methods is still under development. Please refer to the code for not suitably documented sections and complain loudly to the makemaker mailing list.

Not all of the methods below are overridable in a Makefile.PL. Overridable methods are marked as (o). All methods are overridable by a platform specific MM_*.pm file (See the *ExtUtils::MM_VMS* manpage) and the *ExtUtils::MM_OS2* manpage).

**Preloaded methods**

canonpath

No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminated successive slashes and successive "/.".

catdir

Concatenate two or more directory names to form a complete path ending with a directory. But remove the trailing slash from the resulting string, because it doesn't look good, isn't necessary and confuses OS2. Of course, if this is the root directory, don't cut off the trailing slash :−)

catfile

Concatenate one or more directory names and a filename to form a complete path ending with a filename

curdir

Returns a string representing of the current directory. "." on UNIX.

rootdir

Returns a string representing of the root directory. "/" on UNIX.

updir

Returns a string representing of the parent directory. ".." on UNIX.

**SelfLoaded methods**

c_o (o)

Defines the suffix rules to compile different flavors of C files to object files.

cflags (o)

Does very much the same as the cflags script in the perl distribution. It doesn't return the whole compiler command line, but initializes all of its parts. The const_cccmd method then actually returns the definition of the CCCMD macro which uses these parts.

clean (o)
Defines the clean target.

const_cccmd (o)
Returns the full compiler call for C programs and stores the definition in CONST_CCCMD.

const_config (o)
Defines a couple of constants in the Makefile that are imported from `%Config`.

const_loadlibs (o)
Defines EXTRALIBS, LDLOADLIBS, BSLOADLIBS, LD_RUN_PATH. See the *ExtUtils::Liblist* manpage for details.

constants (o)
Initializes lots of constants and .SUFFIXES and .PHONY

depend (o)
Same as macro for the depend attribute.

dir_target (o)
Takes an array of directories that need to exist and returns a Makefile entry for a .exists file in these directories. Returns nothing, if the entry has already been processed. We're helpless though, if the same directory comes as $(FOO) _and_ as "bar". Both of them get an entry, that's why we use "::".

dist (o)
Defines a lot of macros for distribution support.

dist_basics (o)
Defines the targets distclean, distcheck, skipcheck, manifest.

dist_ci (o)
Defines a check in target for RCS.

dist_core (o)
Defeines the targets dist, tardist, zipdist, uutardist, shdist

dist_dir (o)
Defines the scratch directory target that will hold the distribution before tar-ing (or shar-ing).

dist_test (o)
Defines a target that produces the distribution in the scratchdirectory, and runs 'perl Makefile.PL; make ;make test' in that subdirectory.

dlsyms (o)
Used by AIX and VMS to define DL_FUNCS and DL_VARS and write the *.exp files.

dynamic (o)
Defines the dynamic target.

dynamic_bs (o)
Defines targets for bootstrap files.

dynamic_lib (o)
Defines how to produce the *.so (or equivalent) files.

exescan
Deprecated method. Use libscan instead.

extliblist
Called by init_others, and calls ext ExtUtils::Liblist. See the *ExtUtils::Liblist* manpage for details.

file_name_is_absolute
Takes as argument a path and returns true, if it is an absolute path.

find_perl
Finds the executables PERL and FULLPERL

**Methods to actually produce chunks of text for the Makefile**

The methods here are called for each MakeMaker object in the order specified by `@ExtUtils::MakeMaker::MM_Sections`.

fixin
Inserts the sharpbang or equivalent magic number to a script

force (o)
Just writes FORCE:

guess_name
Guess the name of this package by examining the working directory's name. MakeMaker calls this only if the developer has not supplied a NAME attribute.

has_link_code
Returns true if C, XS, MYEXTLIB or similar objects exist within this object that need a compiler. Does not descend into subdirectories as *needs_linking()* does.

init_dirscan
Initializes DIR, XS, PM, C, O_FILES, H, PL_FILES, MAN*PODS, EXE_FILES.

init_main
Initializes NAME, FULLEXT, BASEEXT, PARENT_NAME, DLBASE, PERL_SRC, PERL_LIB, PERL_ARCHLIB, PERL_INC, INSTALLDIRS, INST_*, INSTALL*, PREFIX, CONFIG, AR, AR_STATIC_ARGS, LD, OBJ_EXT, LIB_EXT, EXE_EXT, MAP_TARGET, LIBPERL_A, VERSION_FROM, VERSION, DISTNAME, VERSION_SYM.

init_others
Initializes EXTRALIBS, BSLOADLIBS, LDLOADLIBS, LIBS, LD_RUN_PATH, OBJECT, BOOTDEP, PERLMAINCC, LDFROM, LINKTYPE, NOOP, FIRST_MAKEFILE, MAKEFILE, NOECHO, RM_F, RM_RF, TEST_F, TOUCH, CP, MV, CHMOD, UMASK_NULL

install (o)
Defines the install target.

installbin (o)
Defines targets to install EXE_FILES.

libscan (o)
Takes a path to a file that is found by init_dirscan and returns false if we don't want to include this file in the library. Mainly used to exclude RCS, CVS, and SCCS directories from installation.

linkext (o)
Defines the linkext target which in turn defines the LINKTYPE.

lsdir
Takes as arguments a directory name and a regular expression. Returns all entries in the directory that match the regular expression.

macro (o)
Simple subroutine to insert the macros defined by the macro attribute into the Makefile.

makeaperl (o)
Called by staticmake. Defines how to write the Makefile to produce a static new perl.

By default the Makefile produced includes all the static extensions in the perl library. (Purified versions of library files, e.g., DynaLoader_pure_p1_c0_032.a are automatically ignored to avoid link errors.)

makefile (o)
Defines how to rewrite the Makefile.

manifypods (o)
Defines targets and routines to translate the pods into manpages and put them into the INST_* directories.

maybe_command
Returns true, if the argument is likely to be a command.

maybe_command_in_dirs
method under development. Not yet used. Ask Ilya :–)

needs_linking (o)
Does this module need linking? Looks into subdirectory objects (see also *has_link_code()*)

nicetext
misnamed method (will have to be changed). The MM_Unix method just returns the argument without further processing.

On VMS used to insure that colons marking targets are preceded by space – most Unix Makes don't need this, but it's necessary under VMS to distinguish the target delimiter from a colon appearing as part of a filespec.

parse_version
parse a file and return what you think is `$VERSION` in this file set to

parse_abstract
parse a file and return what you think is the ABSTRACT

pasthru (o)
Defines the string that is passed to recursive make calls in subdirectories.

path
Takes no argument, returns the environment variable PATH as an array.

perl_script
Takes one argument, a file name, and returns the file name, if the argument is likely to be a perl script. On MM_Unix this is true for any ordinary, readable file.

perldepend (o)
Defines the dependency from all *.h files that come with the perl distribution.

ppd
Defines target that creates a PPD (Perl Package Description) file for a binary distribution.

perm_rw (o)
Returns the attribute `PERM_RW` or the string `644`. Used as the string that is passed to the `chmod` command to set the permissions for read/writeable files. MakeMaker chooses `644` because it has turned out in the past that relying on the umask provokes hard-to-track bugreports. When the return value is used by the perl function `chmod`, it is interpreted as an octal value.

perm_rwx (o)
Returns the attribute `PERM_RWX` or the string `755`, i.e. the string that is passed to the `chmod` command to set the permissions for executable files. See also perl_rw.

pm_to_blib
Defines target that copies all files in the hash PM to their destination and autosplits them. See the `DESCRIPTION` entry in the *ExtUtils::Install* manpage

post_constants (o)
Returns an empty string per default. Dedicated to overrides from within Makefile.PL after all constants have been defined.

post_initialize (o)
: Returns an empty string per default. Used in Makefile.PLs to add some chunk of text to the Makefile after the object is initialized.

postamble (o)
: Returns an empty string. Can be used in Makefile.PLs to write some text to the Makefile at the end.

prefixify
: Check a path variable in `$self` from `%Config`, if it contains a prefix, and replace it with another one.

  Takes as arguments an attribute name, a search prefix and a replacement prefix. Changes the attribute in the object.

processPL (o)
: Defines targets to run *.PL files.

realclean (o)
: Defines the realclean target.

replace_manpage_separator
: Takes the name of a package, which may be a nested package, in the form Foo/Bar and replaces the slash with `::`. Returns the replacement.

static (o)
: Defines the static target.

static_lib (o)
: Defines how to produce the *.a (or equivalent) files.

staticmake (o)
: Calls makeaperl.

subdir_x (o)
: Helper subroutine for subdirs

subdirs (o)
: Defines targets to process subdirectories.

test (o)
: Defines the test targets.

test_via_harness (o)
: Helper method to write the test targets

test_via_script (o)
: Other helper method for test.

tool_autosplit (o)
: Defines a simple perl call that runs autosplit. May be deprecated by pm_to_blib soon.

tools_other (o)
: Defines SHELL, LD, TOUCH, CP, MV, RM_F, RM_RF, CHMOD, UMASK_NULL in the Makefile. Also defines the perl programs MKPATH, WARN_IF_OLD_PACKLIST, MOD_INSTALL. DOC_INSTALL, and UNINSTALL.

tool_xsubpp (o)
: Determines typemaps, xsubpp version, prototype behaviour.

top_targets (o)
: Defines the targets all, subdirs, config, and O_FILES

writedoc
: Obsolete, depecated method. Not used since Version 5.21.

xs_c (o)
Defines the suffix rules to compile XS files to C.

xs_o (o)
Defines suffix rules to go from XS to object files directly. This is only intended for broken make implementations.

perl_archive
This is internal method that returns path to libperl.a equivalent to be linked to dynamic extensions. UNIX does not have one but OS2 and Win32 do.

export_list
This is internal method that returns name of a file that is passed to linker to define symbols to be exported. UNIX does not have one but OS2 and Win32 do.

**SEE ALSO**

the *ExtUtils::MakeMaker* manpage

## NAME

ExtUtils::MM_VMS − methods to override UN*X behaviour in ExtUtils::MakeMaker

## SYNOPSIS

```
use ExtUtils::MM_VMS; # Done internally by ExtUtils::MakeMaker if needed
```

## DESCRIPTION

See ExtUtils::MM_Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

### Methods always loaded

eliminate_macros
: Expands MM[KS]/Make macros in a text string, using the contents of identically named elements of `%$self`, and returns the result as a file specification in Unix syntax.

fixpath
: Catchall routine to clean up problem MM[SK]/Make macros. Expands macros in any directory specification, in order to avoid juxtaposing two VMS−syntax directories when MM[SK] is run. Also expands expressions which are all macro, so that we can tell how long the expansion is, and avoid overrunning DCL's command buffer when MM[KS] is running.

  If optional second argument has a TRUE value, then the return string is a VMS−syntax directory specification, if it is FALSE, the return string is a VMS−syntax file specification, and if it is not specified, *fixpath()* checks to see whether it matches the name of a directory in the current default directory, and returns a directory or file specification accordingly.

catdir
: Concatenates a list of file specifications, and returns the result as a VMS−syntax directory specification.

catfile
: Concatenates a list of file specifications, and returns the result as a VMS−syntax directory specification.

wraplist
: Converts a list into a string wrapped at approximately 80 columns.

curdir (override)
: Returns a string representing of the current directory.

rootdir (override)
: Returns a string representing of the root directory.

updir (override)
: Returns a string representing of the parent directory.

### SelfLoaded methods

Those methods which override default MM_Unix methods are marked "(override)", while methods unique to MM_VMS are marked "(specific)". For overridden methods, documentation is limited to an explanation of why this method overrides the MM_Unix method; see the ExtUtils::MM_Unix documentation for more details.

guess_name (override)
: Try to determine name of extension being built. We begin with the name of the current directory. Since VMS filenames are case-insensitive, however, we look for a *.pm* file whose name matches that of the current directory (presumably the 'main' *.pm* file for this extension), and try to find a `package` statement from which to obtain the Mixed::Case package name.

find_perl (override)
: Use VMS file specification syntax and CLI commands to find and invoke Perl images.

path (override)
: Translate logical name DCL$PATH as a searchlist, rather than trying to `split` string value of `$ENV{'PATH'}`.

maybe_command (override)
: Follows VMS naming conventions for executable files. If the name passed in doesn't exactly match an executable file, appends *.Exe* (or equivalent) to check for executable image, and *.Com* to check for DCL procedure. If this fails, checks directories in DCL$PATH and finally *Sys$System:* for an executable file having the name specified, with or without the *.Exe*–equivalent suffix.

maybe_command_in_dirs (override)
: Uses DCL argument quoting on test command line.

perl_script (override)
: If name passed in doesn't specify a readable file, appends *.com* or *.pl* and tries again, since it's customary to have file types on all files under VMS.

file_name_is_absolute (override)
: Checks for VMS directory spec as well as Unix separators.

replace_manpage_separator
: Use as separator a character which is legal in a VMS–syntax file name.

init_others (override)
: Provide VMS–specific forms of various utility commands, then hand off to the default MM_Unix method.

constants (override)
: Fixes up numerous file and directory macros to insure VMS syntax regardless of input syntax. Also adds a few VMS–specific macros and makes lists of files comma-separated.

cflags (override)
: Bypass shell script and produce qualifiers for CC directly (but warn user if a shell script for this extension exists). Fold multiple /Defines into one, since some C compilers pay attention to only one instance of this qualifier on the command line.

const_cccmd (override)
: Adds directives to point C preprocessor to the right place when handling #include <sys/foo.h> directives. Also constructs CC command line a bit differently than MM_Unix method.

pm_to_blib (override)
: DCL *still* accepts a maximum of 255 characters on a command line, so we write the (potentially) long list of file names to a temp file, then persuade Perl to read it instead of the command line to find args.

tool_autosplit (override)
: Use VMS–style quoting on command line.

tool_sxubpp (override)
: Use VMS–style quoting on xsubpp command line.

xsubpp_version (override)
: Test xsubpp exit status according to VMS rules ($sts & 1 ==> good) rather than Unix rules ($sts == 0 ==> good).

tools_other (override)
: Adds a few MM[SK] macros, and shortens some the installatin commands, in order to stay under DCL's 255-character limit. Also changes EQUALIZE_TIMESTAMP to set revision date of target file to one second later than source file, since MMK interprets precisely equal revision dates for a source and target file as a sign that the target needs to be updated.

dist (override)
Provide VMSish defaults for some values, then hand off to default MM_Unix method.

c_o (override)
Use VMS syntax on command line. In particular, $(DEFINE) and $(PERL_INC) have been pulled into $(CCCMD). Also use MM[SK] macros.

xs_c (override)
Use MM[SK] macros.

xs_o (override)
Use MM[SK] macros, and VMS command line for C compiler.

top_targets (override)
Use VMS quoting on command line for Version_check.

dlsyms (override)
Create VMS linker options files specifying universal symbols for this extension's shareable image, and listing other shareable images or libraries to which it should be linked.

dynamic_lib (override)
Use VMS Link command.

dynamic_bs (override)
Use VMS−style quoting on Mkbootstrap command line.

static_lib (override)
Use VMS commands to manipulate object library.

manifypods (override)
Use VMS−style quoting on command line, and VMS logical name to specify fallback location at build time if we can't find pod2man.

processPL (override)
Use VMS−style quoting on command line.

installbin (override)
Stay under DCL's 255 character command line limit once again by splitting potentially long list of files across multiple lines in `realclean` target.

subdir_x (override)
Use VMS commands to change default directory.

clean (override)
Split potentially long list of files across multiple commands (in order to stay under the magic command line limit). Also use MM[SK] commands for handling subdirectories.

realclean (override)
Guess what we're working around? Also, use MM[SK] for subdirectories.

dist_basics (override)
Use VMS−style quoting on command line.

dist_core (override)
Syntax for invoking *VMS_Share* differs from that for Unix *shar*, so `shdist` target actions are VMS−specific.

dist_dir (override)
Use VMS−style quoting on command line.

dist_test (override)
Use VMS commands to change default directory, and use VMS−style quoting on command line.

install (override)
: Work around DCL's 255 character limit several times,and use VMS−style command line quoting in a few cases.

perldepend (override)
: Use VMS−style syntax for files; it's cheaper to just do it directly here than to have the MM_Unix method call `catfile` repeatedly. Also, if we have to rebuild Config.pm, use MM[SK] to do it.

makefile (override)
: Use VMS commands and quoting.

test (override)
: Use VMS commands for handling subdirectories.

test_via_harness (override)
: Use VMS−style quoting on command line.

test_via_script (override)
: Use VMS−style quoting on command line.

makeaperl (override)
: Undertake to build a new set of Perl images using VMS commands. Since VMS does dynamic loading, it's not necessary to statically link each extension into the Perl image, so this isn't the normal build path. Consequently, it hasn't really been tested, and may well be incomplete.

nicetext (override)
: Insure that colons marking targets are preceded by space, in order to distinguish the target delimiter from a colon appearing as part of a filespec.

**NAME**

ExtUtils::MM_Win32 − methods to override UN*X behaviour in ExtUtils::MakeMaker

**SYNOPSIS**

```
use ExtUtils::MM_Win32; # Done internally by ExtUtils::MakeMaker if needed
```

**DESCRIPTION**

See ExtUtils::MM_Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

catfile
: Concatenate one or more directory names and a filename to form a complete path ending with a filename

constants (o)
: Initializes lots of constants and .SUFFIXES and .PHONY

static_lib (o)
: Defines how to produce the *.a (or equivalent) files.

dynamic_bs (o)
: Defines targets for bootstrap files.

dynamic_lib (o)
: Defines how to produce the *.so (or equivalent) files.

canonpath
: No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminated successive slashes and successive "/.".

perl_script
: Takes one argument, a file name, and returns the file name, if the argument is likely to be a perl script. On MM_Unix this is true for any ordinary, readable file.

pm_to_blib
: Defines target that copies all files in the hash PM to their destination and autosplits them. See the `DESCRIPTION` entry in the *ExtUtils::Install* manpage

test_via_harness (o)
: Helper method to write the test targets

tool_autosplit (override)
: Use Win32 quoting on command line.

tools_other (o)
: Win32 overrides.

  Defines SHELL, LD, TOUCH, CP, MV, RM_F, RM_RF, CHMOD, UMASK_NULL in the Makefile. Also defines the perl programs MKPATH, WARN_IF_OLD_PACKLIST, MOD_INSTALL. DOC_INSTALL, and UNINSTALL.

xs_o (o)
: Defines suffix rules to go from XS to object files directly. This is only intended for broken make implementations.

top_targets (o)
: Defines the targets all, subdirs, config, and O_FILES

manifypods (o)
: We don't want manpage process. XXX add pod2html support later.

dist_ci (o)
: Same as MM_Unix version (changes command-line quoting).

dist_core (o)
: Same as MM_Unix version (changes command-line quoting).

pasthru (o)
: Defines the string that is passed to recursive make calls in subdirectories.

**NAME**

ExtUtils::Packlist − manage .packlist files

**SYNOPSIS**

```
use ExtUtils::Packlist;
my ($pl) = ExtUtils::Packlist->new('.packlist');
$pl->read('/an/old/.packlist');
my @missing_files = $pl->validate();
$pl->write('/a/new/.packlist');

$pl->{'/some/file/name'}++;
   or
$pl->{'/some/other/file/name'} = { type => 'file',
                                    from => '/some/file' };
```

**DESCRIPTION**

ExtUtils::Packlist provides a standard way to manage .packlist files. Functions are provided to read and write .packlist files. The original .packlist format is a simple list of absolute pathnames, one per line. In addition, this package supports an extended format, where as well as a filename each line may contain a list of attributes in the form of a space separated list of key=value pairs. This is used by the installperl script to differentiate between files and links, for example.

**USAGE**

The hash reference returned by the *new()* function can be used to examine and modify the contents of the .packlist. Items may be added/deleted from the .packlist by modifying the hash. If the value associated with a hash key is a scalar, the entry written to the .packlist by any subsequent *write()* will be a simple filename. If the value is a hash, the entry written will be the filename followed by the key=value pairs from the hash. Reading back the .packlist will recreate the original entries.

**FUNCTIONS**

new()

This takes an optional parameter, the name of a .packlist. If the file exists, it will be opened and the contents of the file will be read. The *new()* method returns a reference to a hash. This hash holds an entry for each line in the .packlist. In the case of old-style .packlists, the value associated with each key is undef. In the case of new-style .packlists, the value associated with each key is a hash containing the key=value pairs following the filename in the .packlist.

read()

This takes an optional parameter, the name of the .packlist to be read. If no file is specified, the .packlist specified to *new()* will be read. If the .packlist does not exist, Carp::croak will be called.

write()

This takes an optional parameter, the name of the .packlist to be written. If no file is specified, the .packlist specified to *new()* will be overwritten.

validate()

This checks that every file listed in the .packlist actually exists. If an argument which evaluates to true is given, any missing files will be removed from the internal hash. The return value is a list of the missing files, which will be empty if they all exist.

packlist_file()

This returns the name of the associated .packlist file

**EXAMPLE**

Here's `modrm`, a little utility to cleanly remove an installed module.

```
#!/usr/local/bin/perl -w
```

```
use strict;
use IO::Dir;
use ExtUtils::Packlist;
use ExtUtils::Installed;

sub emptydir($) {
    my ($dir) = @_;
    my $dh = IO::Dir->new($dir) || return(0);
    my @count = $dh->read();
    $dh->close();
    return(@count == 2 ? 1 : 0);
}

# Find all the installed packages
print("Finding all installed modules...\n");
my $installed = ExtUtils::Installed->new();

foreach my $module (grep(!/^Perl$/, $installed->modules())) {
   my $version = $installed->version($module) || "???";
   print("Found module $module Version $version\n");
   print("Do you want to delete $module? [n] ");
   my $r = <STDIN>; chomp($r);
   if ($r && $r =~ /^y/i) {
      # Remove all the files
      foreach my $file (sort($installed->files($module))) {
         print("rm $file\n");
         unlink($file);
      }
      my $pf = $installed->packlist($module)->packlist_file();
      print("rm $pf\n");
      unlink($pf);
      foreach my $dir (sort($installed->directory_tree($module))) {
         if (emptydir($dir)) {
            print("rmdir $dir\n");
            rmdir($dir);
         }
      }
   }
}
```

**AUTHOR**

Alan Burlison <Alan.Burlison@uk.sun.com>

**NAME**

ExtUtils::testlib – add blib/* directories to `@INC`

**SYNOPSIS**

```
use ExtUtils::testlib;
```

**DESCRIPTION**

After an extension has been built and before it is installed it may be desirable to test it bypassing `make test`. By adding

```
use ExtUtils::testlib;
```

to a test program the intermediate directories used by `make` are added to `@INC`.

## NAME

Fatal – replace functions with equivalents which succeed or die

## SYNOPSIS

```
use Fatal qw(open close);

sub juggle { . . . }
import Fatal 'juggle';
```

## DESCRIPTION

`Fatal` provides a way to conveniently replace functions which normally return a false value when they fail with equivalents which halt execution if they are not successful. This lets you use these functions without having to test their return values explicitly on each call. Errors are reported via `die`, so you can trap them using `$SIG{__DIE__}` if you wish to take some action before the program exits.

The do-or-die equivalents are set up simply by calling Fatal's `import` routine, passing it the names of the functions to be replaced. You may wrap both user-defined functions and overridable CORE operators (except `exec`, `system` which cannot be expressed via prototypes) in this way.

## AUTHOR

Lionel.Cons@cern.ch

prototype updates by Ilya Zakharevich ilya@math.ohio-state.edu

## NAME

Fcntl − load the C Fcntl.h defines

## SYNOPSIS

```
use Fcntl;
use Fcntl qw(:DEFAULT :flock);
```

## DESCRIPTION

This module is just a translation of the C *fnctl.h* file. Unlike the old mechanism of requiring a translated *fnctl.ph* file, this uses the **h2xs** program (see the Perl source distribution) and your native C compiler. This means that it has a far more likely chance of getting the numbers right.

## NOTE

Only `#define` symbols get translated; you must still correctly pack up your own arguments to pass as args for locking functions, etc.

## EXPORTED SYMBOLS

By default your system's F_* and O_* constants (eg, F_DUPFD and O_CREAT) and the FD_CLOEXEC constant are exported into your namespace.

You can request that the *flock()* constants (LOCK_SH, LOCK_EX, LOCK_NB and LOCK_UN) be provided by using the tag `:flock`. See the *Exporter* manpage.

You can request that the old constants (FAPPEND, FASYNC, FCREAT, FDEFER, FEXCL, FNDELAY, FNONBLOCK, FSYNC, FTRUNC) be provided for compatibility reasons by using the tag `:Fcompat`. For new applications the newer versions of these constants are suggested (O_APPEND, O_ASYNC, O_CREAT, O_DEFER, O_EXCL, O_NDELAY, O_NONBLOCK, O_SYNC, O_TRUNC).

Please refer to your native *fcntl()* and *open()* documentation to see what constants are implemented in your system.

## NAME

fileparse − split a pathname into pieces

basename − extract just the filename from a path

dirname − extract just the directory from a path

## SYNOPSIS

```
use File::Basename;

($name,$path,$suffix) = fileparse($fullname,@suffixlist)
fileparse_set_fstype($os_string);
$basename = basename($fullname,@suffixlist);
$dirname = dirname($fullname);

($name,$path,$suffix) = fileparse("lib/File/Basename.pm","\.pm");
fileparse_set_fstype("VMS");
$basename = basename("lib/File/Basename.pm",".pm");
$dirname = dirname("lib/File/Basename.pm");
```

## DESCRIPTION

These routines allow you to parse file specifications into useful pieces using the syntax of different operating systems.

fileparse_set_fstype

You select the syntax via the routine *fileparse_set_fstype()*.

If the argument passed to it contains one of the substrings "VMS", "MSDOS", "MacOS", "AmigaOS" or "MSWin32", the file specification syntax of that operating system is used in future calls to *fileparse()*, *basename()*, and *dirname()*. If it contains none of these substrings, UNIX syntax is used. This pattern matching is case-insensitive. If you've selected VMS syntax, and the file specification you pass to one of these routines contains a "/", they assume you are using UNIX emulation and apply the UNIX syntax rules instead, for that function call only.

If the argument passed to it contains one of the substrings "VMS", "MSDOS", "MacOS", "AmigaOS", "os2", "MSWin32" or "RISCOS", then the pattern matching for suffix removal is performed without regard for case, since those systems are not case-sensitive when opening existing files (though some of them preserve case on file creation).

If you haven't called *fileparse_set_fstype()*, the syntax is chosen by examining the builtin variable `$^O` according to these rules.

fileparse

The *fileparse()* routine divides a file specification into three parts: a leading **path**, a file **name**, and a **suffix**. The **path** contains everything up to and including the last directory separator in the input file specification. The remainder of the input file specification is then divided into **name** and **suffix** based on the optional patterns you specify in `@suffixlist`. Each element of this list is interpreted as a regular expression, and is matched against the end of **name**. If this succeeds, the matching portion of **name** is removed and prepended to **suffix**. By proper use of `@suffixlist`, you can remove file types or versions for examination.

You are guaranteed that if you concatenate **path**, **name**, and **suffix** together in that order, the result will denote the same file as the input file specification.

## EXAMPLES

Using UNIX file syntax:

```
($base,$path,$type) = fileparse('/virgil/aeneid/draft.book7',
                                 '\.book\d+');
```

would yield

```
$base eq 'draft'
$path eq '/virgil/aeneid/',
$type eq '.book7'
```

Similarly, using VMS syntax:

```
($name,$dir,$type) = fileparse('Doc_Root:[Help]Rhetoric.Rnh',
                                '\..*');
```

would yield

```
$name eq 'Rhetoric'
$dir  eq 'Doc_Root:[Help]'
$type eq '.Rnh'
```

`basename`

The *basename()* routine returns the first element of the list produced by calling *fileparse()* with the same arguments, except that it always quotes metacharacters in the given suffixes. It is provided for programmer compatibility with the UNIX shell command *basename* (1).

`dirname`

The *dirname()* routine returns the directory portion of the input file specification. When using VMS or MacOS syntax, this is identical to the second element of the list produced by calling *fileparse()* with the same input file specification. (Under VMS, if there is no directory information in the input file specification, then the current default device and directory are returned.) When using UNIX or MSDOS syntax, the return value conforms to the behavior of the UNIX shell command *dirname* (1). This is usually the same as the behavior of *fileparse()*, but differs in some cases. For example, for the input file specification *lib/*, *fileparse()* considers the directory name to be *lib/*, while *dirname()* considers the directory name to be .).

**NAME**

validate − run many filetest checks on a tree

**SYNOPSIS**

```
use File::CheckTree;

$warnings += validate( q{
    /vmunix                 -e || die
    /boot                   -e || die
    /bin                    cd
        csh                 -ex
        csh                 !-ug
        sh                  -ex
        sh                  !-ug
    /usr                    -d || warn "What happened to $file?\n"
});
```

**DESCRIPTION**

The *validate()* routine takes a single multiline string consisting of lines containing a filename plus a file test to try on it. (The file test may also be a "cd", causing subsequent relative filenames to be interpreted relative to that directory.) After the file test you may put `|| die` to make it a fatal error if the file test fails. The default is `|| warn`. The file test may optionally have a "!' prepended to test for the opposite condition. If you do a cd and then list some relative filenames, you may want to indent them slightly for readability. If you supply your own *die()* or *warn()* message, you can use `$file` to interpolate the filename.

Filetests may be bunched: "−rwx" tests for all of `-r`, `-w`, and `-x`. Only the first failed test of the bunch will produce a warning.

The routine returns the number of warnings issued.

**NAME**

File::Compare − Compare files or filehandles

**SYNOPSIS**

```
use File::Compare;

if (compare("file1","file2") == 0) {
    print "They're equal\n";
}
```

**DESCRIPTION**

The File::Compare::compare function compares the contents of two sources, each of which can be a file or a file handle. It is exported from File::Compare by default.

File::Compare::cmp is a synonym for File::Compare::compare. It is exported from File::Compare only by request.

**RETURN**

File::Compare::compare return 0 if the files are equal, 1 if the files are unequal, or −1 if an error was encountered.

**AUTHOR**

File::Compare was written by Nick Ing-Simmons. Its original documentation was written by Chip Salzenberg.

**NAME**

File::Copy – Copy files or filehandles

**SYNOPSIS**

```
use File::Copy;

copy("file1","file2");
copy("Copy.pm",\*STDOUT);'
move("/dev1/fileA","/dev2/fileB");

use POSIX;
use File::Copy cp;

$n=FileHandle->new("/dev/null","r");
cp($n,"x");'
```

**DESCRIPTION**

The File::Copy module provides two basic functions, `copy` and `move`, which are useful for getting the contents of a file from one place to another.

- The `copy` function takes two parameters: a file to copy from and a file to copy to. Either argument may be a string, a FileHandle reference or a FileHandle glob. Obviously, if the first argument is a filehandle of some sort, it will be read from, and if it is a file *name* it will be opened for reading. Likewise, the second argument will be written to (and created if need be).

  **Note that passing in files as handles instead of names may lead to loss of information on some operating systems; it is recommended that you use file names whenever possible.** Files are opened in binary mode where applicable. To get a consistent behavour when copying from a filehandle to a file, use `binmode` on the filehandle.

  An optional third parameter can be used to specify the buffer size used for copying. This is the number of bytes from the first file, that wil be held in memory at any given time, before being written to the second file. The default buffer size depends upon the file, but will generally be the whole file (up to 2Mb), or 1k for filehandles that do not reference files (eg. sockets).

  You may use the syntax `use File::Copy "cp"` to get at the "cp" alias for this function. The syntax is *exactly* the same.

- The `move` function also takes two parameters: the current name and the intended name of the file to be moved. If the destination already exists and is a directory, and the source is not a directory, then the source file will be renamed into the directory specified by the destination.

  If possible, *move()* will simply rename the file. Otherwise, it copies the file to the new location and deletes the original. If an error occurs during this copy-and-delete process, you may be left with a (possibly partial) copy of the file under the destination name.

  You may use the "mv" alias for this function in the same way that you may use the "cp" alias for `copy`.

File::Copy also provides the `syscopy` routine, which copies the file specified in the first parameter to the file specified in the second parameter, preserving OS–specific attributes and file structure. For Unix systems, this is equivalent to the simple `copy` routine. For VMS systems, this calls the `rmscopy` routine (see below). For OS/2 systems, this calls the `syscopy` XSUB directly.

**Special behavior if** `syscopy` is defined (VMS and OS/2)

If both arguments to `copy` are not file handles, then `copy` will perform a "system copy" of the input file to a new output file, in order to preserve file attributes, indexed file structure, *etc.* The buffer size parameter is ignored. If either argument to `copy` is a handle to an opened file, then data is copied using Perl operators, and no effort is made to preserve file attributes or record structure.

The system copy routine may also be called directly under VMS and OS/2 as `File::Copy::syscopy` (or under VMS as `File::Copy::rmscopy`, which is the routine that does the actual work for syscopy).

rmscopy($from,$to[,$date_flag])

The first and second arguments may be strings, typeglobs, typeglob references, or objects inheriting from IO::Handle; they are used in all cases to obtain the *filespec* of the input and output files, respectively. The name and type of the input file are used as defaults for the output file, if necessary.

A new version of the output file is always created, which inherits the structure and RMS attributes of the input file, except for owner and protections (and possibly timestamps; see below). All data from the input file is copied to the output file; if either of the first two parameters to `rmscopy` is a file handle, its position is unchanged. (Note that this means a file handle pointing to the output file will be associated with an old version of that file after `rmscopy` returns, not the newly created version.)

The third parameter is an integer flag, which tells `rmscopy` how to handle timestamps. If it is < 0, none of the input file's timestamps are propagated to the output file. If it is > 0, then it is interpreted as a bitmask: if bit 0 (the LSB) is set, then timestamps other than the revision date are propagated; if bit 1 is set, the revision date is propagated. If the third parameter to `rmscopy` is 0, then it behaves much like the DCL COPY command: if the name or type of the output file was explicitly specified, then no timestamps are propagated, but if they were taken implicitly from the input filespec, then all timestamps other than the revision date are propagated. If this parameter is not supplied, it defaults to 0.

Like `copy`, `rmscopy` returns 1 on success. If an error occurs, it sets `$!`, deletes the output file, and returns 0.

**RETURN**

All functions return 1 on success, 0 on failure. $! will be set if an error was encountered.

**AUTHOR**

File::Copy was written by Aaron Sherman *<ajs@ajs.com>* in 1995, and updated by Charles Bailey *<bailey@genetics.upenn.edu>* in 1996.

## NAME

File::DosGlob – DOS like globbing and then some

## SYNOPSIS

```
require 5.004;

# override CORE::glob in current package
use File::DosGlob 'glob';

# override CORE::glob in ALL packages (use with extreme caution!)
use File::DosGlob 'GLOBAL_glob';

@perlfiles = glob  "..\\pe?l/*.p?";
print <..\\pe?l/*.p?>;

# from the command line (overrides only in main::)
> perl -MFile::DosGlob=glob -e "print <../pe*/*p?>"
```

## DESCRIPTION

A module that implements DOS–like globbing with a few enhancements. It is largely compatible with perlglob.exe (the M$ setargv.obj version) in all but one respect— it understands wildcards in directory components.

For example, `<..\\l*b\\file/*glob.p?>` will work as expected (in that it will find something like '..\lib\File/DosGlob.pm' alright). Note that all path components are case-insensitive, and that backslashes and forward slashes are both accepted, and preserved. You may have to double the backslashes if you are putting them in literally, due to double-quotish parsing of the pattern by perl.

Extending it to csh patterns is left as an exercise to the reader.

## EXPORTS (by request only)

*glob()*

## BUGS

Should probably be built into the core, and needs to stop pandering to DOS habits. Needs a dose of optimizium too.

## AUTHOR

Gurusamy Sarathy <gsar@umich.edu>

## HISTORY

- Support for globally overriding *glob()* (GSAR 3-JUN–98)
- Scalar context, independent iterator context fixes (GSAR 15-SEP–97)
- A few dir-vs-file optimizations result in glob importation being 10 times faster than using perlglob.exe, and using perlglob.bat is only twice as slow as perlglob.exe (GSAR 28-MAY–97)
- Several cleanups prompted by lack of compatible perlglob.exe under Borland (GSAR 27-MAY–97)
- Initial version (GSAR 20-FEB–97)

## SEE ALSO

perl

perlglob.bat

**NAME**

find − traverse a file tree

finddepth − traverse a directory structure depth-first

**SYNOPSIS**

```
use File::Find;
find(\&wanted, '/foo','/bar');
sub wanted { ... }

use File::Find;
finddepth(\&wanted, '/foo','/bar');
sub wanted { ... }
```

**DESCRIPTION**

The first argument to *find()* is either a hash reference describing the operations to be performed for each file, or a code reference. If it is a hash reference, then the value for the key `wanted` should be a code reference. This code reference is called *the wanted() function* below.

Currently the only other supported key for the above hash is `bydepth`, in presense of which the walk over directories is performed depth-first. Entry point *finddepth()* is a shortcut for specifying `{ bydepth = 1}`> in the first argument of *find()*.

The *wanted()* function does whatever verifications you want. `$File::Find::dir` contains the current directory name, and `$_` the current filename within that directory. `$File::Find::name` contains `"$File::Find::dir/$_"`. You are *chdir()*'d to `$File::Find::dir` when the function is called. The function may set `$File::Find::prune` to prune the tree.

File::Find assumes that you don't alter the `$_` variable. If you do then make sure you return it to its original value before exiting your function.

This library is useful for the `find2perl` tool, which when fed,

```
find2perl / -name .nfs\* -mtime +7 \
    -exec rm -f {} \; -o -fstype nfs -prune
```

produces something like:

```
sub wanted {
    /^\.nfs.*$/ &&
    (($dev,$ino,$mode,$nlink,$uid,$gid) = lstat($_)) &&
    int(-M _) > 7 &&
    unlink($_)
    ||
    ($nlink || (($dev,$ino,$mode,$nlink,$uid,$gid) = lstat($_))) &&
    $dev < 0 &&
    ($File::Find::prune = 1);
}
```

Set the variable `$File::Find::dont_use_nlink` if you're using AFS, since AFS cheats.

`finddepth` is just like `find`, except that it does a depth-first search.

Here's another interesting wanted function. It will find all symlinks that don't resolve:

```
sub wanted {
    -l && !-e && print "bogus link: $File::Find::name\n";
}
```

**BUGS**

There is no way to make find or finddepth follow symlinks.

## NAME

File::Path − create or remove a series of directories

## SYNOPSIS

```
use File::Path

mkpath(['/foo/bar/baz', 'blurfl/quux'], 1, 0711);

rmtree(['foo/bar/baz', 'blurfl/quux'], 1, 1);
```

## DESCRIPTION

The `mkpath` function provides a convenient way to create directories, even if your `mkdir` kernel call won't create more than one level of directory at a time. `mkpath` takes three arguments:

- the name of the path to create, or a reference to a list of paths to create,
- a boolean value, which if TRUE will cause `mkpath` to print the name of each directory as it is created (defaults to FALSE), and
- the numeric mode to use when creating the directories (defaults to 0777)

It returns a list of all directories (including intermediates, determined using the Unix '/' separator) created.

Similarly, the `rmtree` function provides a convenient way to delete a subtree from the directory structure, much like the Unix command `rm -r`. `rmtree` takes three arguments:

- the root of the subtree to delete, or a reference to a list of roots. All of the files and directories below each root, as well as the roots themselves, will be deleted.
- a boolean value, which if TRUE will cause `rmtree` to print a message each time it examines a file, giving the name of the file, and indicating whether it's using `rmdir` or `unlink` to remove it, or that it's skipping it. (defaults to FALSE)
- a boolean value, which if TRUE will cause `rmtree` to skip any files to which you do not have delete access (if running under VMS) or write access (if running under another OS). This will change in the future when a criterion for 'delete permission' under OSs other than VMS is settled. (defaults to FALSE)

It returns the number of files successfully deleted. Symlinks are treated as ordinary files.

**NOTE:** If the third parameter is not TRUE, `rmtree` is **unsecure** in the face of failure or interruption. Files and directories which were not deleted may be left with permissions reset to allow world read and write access. Note also that the occurrence of errors in rmtree can be determined *only* by trapping diagnostic messages using `$SIG{__WARN__}`; it is not apparent from the return value. Therefore, you must be extremely careful about using `rmtree($foo,$bar,0` in situations where security is an issue.

## AUTHORS

Tim Bunce <*Tim.Bunce@ig.co.uk*> and Charles Bailey <*bailey@genetics.upenn.edu*>

## REVISION

Current `$VERSION` is 1.0401.

## NAME

File::Spec – portably perform operations on file names

## SYNOPSIS

```
use File::Spec;

$x=File::Spec->catfile('a','b','c');
```

which returns 'a/b/c' under Unix.

## DESCRIPTION

This module is designed to support operations commonly performed on file specifications (usually called "file names", but not to be confused with the contents of a file, or Perl's file handles), such as concatenating several directory and file names into a single path, or determining whether a path is rooted. It is based on code directly taken from MakeMaker 5.17, code written by Andreas König, Andy Dougherty, Charles Bailey, Ilya Zakharevich, Paul Schinder, and others.

Since these functions are different for most operating systems, each set of OS specific routines is available in a separate module, including:

```
File::Spec::Unix
File::Spec::Mac
File::Spec::OS2
File::Spec::Win32
File::Spec::VMS
```

The module appropriate for the current OS is automatically loaded by File::Spec. Since some modules (like VMS) make use of OS specific facilities, it may not be possible to load all modules under all operating systems.

Since File::Spec is object oriented, subroutines should not called directly, as in:

```
File::Spec::catfile('a','b');
```

```
but rather as class methods:

    File::Spec->catfile('a','b');
```

For a reference of available functions, pleaes consult the *File::Spec::Unix* manpage, which contains the entire set, and inherited by the modules for other platforms. For further information, please see the *File::Spec::Mac* manpage, the *File::Spec::OS2* manpage, the *File::Spec::Win32* manpage, or the *File::Spec::VMS* manpage.

## SEE ALSO

File::Spec::Unix, File::Spec::Mac, File::Spec::OS2, File::Spec::Win32, File::Spec::VMS, ExtUtils::MakeMaker

## AUTHORS

Kenneth Albanowski <*kjahds@kjahds.com*>, Andy Dougherty <*doughera@lafcol.lafayette.edu*>, Andreas König <*A.Koenig@franz.ww.TU−Berlin.DE*>, Tim Bunce <*Tim.Bunce@ig.co.uk*>. VMS support by Charles Bailey <*bailey@genetics.upenn.edu*>. OS/2 support by Ilya Zakharevich <*ilya@math.ohio-state.edu*>. Mac support by Paul Schinder <*schinder@pobox.com*>.

## NAME

File::Spec::Mac – File::Spec for MacOS

## SYNOPSIS

```
require File::Spec::Mac;
```

## DESCRIPTION

Methods for manipulating file specifications.

## METHODS

canonpath

On MacOS, there's nothing to be done. Returns what it's given.

catdir

Concatenate two or more directory names to form a complete path ending with a directory. Put a trailing : on the end of the complete path if there isn't one, because that's what's done in MacPerl's environment.

The fundamental requirement of this routine is that

```
File::Spec->catdir(split(":",$path)) eq $path
```

But because of the nature of Macintosh paths, some additional possibilities are allowed to make using this routine give resonable results for some common situations. Here are the rules that are used. Each argument has its trailing ":" removed. Each argument, except the first, has its leading ":" removed. They are then joined together by a ":".

So

```
File::Spec->catdir("a","b") = "a:b:"
File::Spec->catdir("a:",":b") = "a:b:"
File::Spec->catdir("a:","b") = "a:b:"
File::Spec->catdir("a",":b") = "a:b"
File::Spec->catdir("a","","b") = "a::b"
```

etc.

To get a relative path (one beginning with :), begin the first argument with : or put a "" as the first argument.

If you don't want to worry about these rules, never allow a ":" on the ends of any of the arguments except at the beginning of the first.

Under MacPerl, there is an additional ambiguity. Does the user intend that

```
File::Spec->catfile("LWP","Protocol","http.pm")
```

be relative or absolute? There's no way of telling except by checking for the existance of LWP: or :LWP, and even there he may mean a dismounted volume or a relative path in a different directory (like in `@INC`). So those checks aren't done here. This routine will treat this as absolute.

catfile

Concatenate one or more directory names and a filename to form a complete path ending with a filename. Since this uses catdir, the same caveats apply. Note that the leading : is removed from the filename, so that

```
File::Spec->catfile($ENV{HOME},"file");
```

and

```
File::Spec->catfile($ENV{HOME},":file");
```

give the same answer, as one might expect.

curdir
: Returns a string representing of the current directory.

rootdir
: Returns a string representing the root directory. Under MacPerl, returns the name of the startup volume, since that's the closest in concept, although other volumes aren't rooted there. On any other platform returns '', since there's no common way to indicate "root directory" across all Macs.

updir
: Returns a string representing the parent directory.

file_name_is_absolute
: Takes as argument a path and returns true, if it is an absolute path. In the case where a name can be either relative or absolute (for example, a folder named "HD" in the current working directory on a drive named "HD"), relative wins. Use ":" in the appropriate place in the path if you want to distinguish unambiguously.

path
: Returns the null list for the MacPerl application, since the concept is usually meaningless under MacOS. But if you're using the MacPerl tool under MPW, it gives back `$ENV{Commands}` suitably split, as is done in :lib:ExtUtils:MM_Mac.pm.

**SEE ALSO**

the *File::Spec* manpage

**NAME**

File::Spec::OS2 – methods for OS/2 file specs

**SYNOPSIS**

```
use File::Spec::OS2; # Done internally by File::Spec if needed
```

**DESCRIPTION**

See File::Spec::Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

**NAME**

File::Spec::Unix − methods used by File::Spec

**SYNOPSIS**

```
require File::Spec::Unix;
```

**DESCRIPTION**

Methods for manipulating file specifications.

**METHODS**

canonpath
: No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminated successive slashes and successive "/.".

catdir
: Concatenate two or more directory names to form a complete path ending with a directory. But remove the trailing slash from the resulting string, because it doesn't look good, isn't necessary and confuses OS2. Of course, if this is the root directory, don't cut off the trailing slash :−)

catfile
: Concatenate one or more directory names and a filename to form a complete path ending with a filename

curdir
: Returns a string representing of the current directory. "." on UNIX.

rootdir
: Returns a string representing of the root directory. "/" on UNIX.

updir
: Returns a string representing of the parent directory. ".." on UNIX.

no_upwards
: Given a list of file names, strip out those that refer to a parent directory. (Does not strip symlinks, only '.', '..', and equivalents.)

file_name_is_absolute
: Takes as argument a path and returns true, if it is an absolute path.

path
: Takes no argument, returns the environment variable PATH as an array.

join
: join is the same as catfile.

nativename
: TBW.

**SEE ALSO**

the *File::Spec* manpage

## NAME

File::Spec::VMS − methods for VMS file specs

## SYNOPSIS

```
use File::Spec::VMS; # Done internally by File::Spec if needed
```

## DESCRIPTION

See File::Spec::Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

### Methods always loaded

catdir
: Concatenates a list of file specifications, and returns the result as a VMS−syntax directory specification.

catfile
: Concatenates a list of file specifications, and returns the result as a VMS−syntax directory specification.

curdir (override)
: Returns a string representing of the current directory.

rootdir (override)
: Returns a string representing of the root directory.

updir (override)
: Returns a string representing of the parent directory.

path (override)
: Translate logical name DCL$PATH as a searchlist, rather than trying to `split` string value of `$ENV{'PATH'}`.

file_name_is_absolute (override)
: Checks for VMS directory spec as well as Unix separators.

## NAME

File::Spec::Win32 – methods for Win32 file specs

## SYNOPSIS

```
use File::Spec::Win32; # Done internally by File::Spec if needed
```

## DESCRIPTION

See File::Spec::Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

catfile

Concatenate one or more directory names and a filename to form a complete path ending with a filename

canonpath

No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminated successive slashes and successive "/.".

## NAME

File::stat − by-name interface to Perl's built-in *stat()* functions

## SYNOPSIS

```
use File::stat;
$st = stat($file) or die "No $file: $!";
if ( ($st->mode & 0111) && $st->nlink > 1) ) {
    print "$file is executable with lotsa links\n";
}

use File::stat qw(:FIELDS);
stat($file) or die "No $file: $!";
if ( ($st_mode & 0111) && $st_nlink > 1) ) {
    print "$file is executable with lotsa links\n";
}
```

## DESCRIPTION

This module's default exports override the core *stat()* and *lstat()* functions, replacing them with versions that return "File::stat" objects. This object has methods that return the similarly named structure field name from the *stat*(2) function; namely, dev, ino, mode, nlink, uid, gid, rdev, size, atime, mtime, ctime, blksize, and blocks.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your *stat()* and *lstat()* functions.) Access these fields as variables named with a preceding `st_` in front their method names. Thus, `$stat_obj->dev()` corresponds to `$st_dev` if you import the fields.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

## NOTE

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

## AUTHOR

Tom Christiansen

**NAME**

FileCache − keep more files open than the system permits

**SYNOPSIS**

```
cacheout $path;
print $path @data;
```

**DESCRIPTION**

The `cacheout` function will make sure that there's a filehandle open for writing available as the pathname you give it. It automatically closes and re-opens files if you exceed your system file descriptor maximum.

**BUGS**

*sys/param.h* lies with its `NOFILE` define on some systems, so you may have to set `$FileCache::cacheout_maxopen` yourself.

## NAME

FileHandle – supply object methods for filehandles

## SYNOPSIS

```
use FileHandle;

$fh = new FileHandle;
if ($fh->open("< file")) {
    print <$fh>;
    $fh->close;
}

$fh = new FileHandle "> FOO";
if (defined $fh) {
    print $fh "bar\n";
    $fh->close;
}

$fh = new FileHandle "file", "r";
if (defined $fh) {
    print <$fh>;
    undef $fh;       # automatically closes the file
}

$fh = new FileHandle "file", O_WRONLY|O_APPEND;
if (defined $fh) {
    print $fh "corge\n";
    undef $fh;       # automatically closes the file
}

$pos = $fh->getpos;
$fh->setpos($pos);

$fh->setvbuf($buffer_var, _IOLBF, 1024);

($readfh, $writefh) = FileHandle::pipe;

autoflush STDOUT 1;
```

## DESCRIPTION

NOTE: This class is now a front-end to the IO::* classes.

`FileHandle::new` creates a `FileHandle`, which is a reference to a newly created symbol (see the `Symbol` package). If it receives any parameters, they are passed to `FileHandle::open`; if the open fails, the `FileHandle` object is destroyed. Otherwise, it is returned to the caller.

`FileHandle::new_from_fd` creates a `FileHandle` like `new` does. It requires two parameters, which are passed to `FileHandle::fdopen`; if the fdopen fails, the `FileHandle` object is destroyed. Otherwise, it is returned to the caller.

`FileHandle::open` accepts one parameter or two. With one parameter, it is just a front end for the built-in `open` function. With two parameters, the first parameter is a filename that may include whitespace or other special characters, and the second parameter is the open mode, optionally followed by a file permission value.

If `FileHandle::open` receives a Perl mode string (">", "+<", etc.) or a POSIX *fopen()* mode string ("w", "r+", etc.), it uses the basic Perl `open` operator.

If `FileHandle::open` is given a numeric mode, it passes that mode and the optional permissions value to the Perl `sysopen` operator. For convenience, `FileHandle::import` tries to import the O_XXX constants from the Fcntl module. If dynamic loading is not available, this may fail, but the rest of FileHandle will still work.

`FileHandle::fdopen` is like `open` except that its first parameter is not a filename but rather a file handle name, a FileHandle object, or a file descriptor number.

If the C functions *fgetpos()* and *fsetpos()* are available, then `FileHandle::getpos` returns an opaque value that represents the current position of the FileHandle, and `FileHandle::setpos` uses that value to return to a previously visited position.

If the C function *setvbuf()* is available, then `FileHandle::setvbuf` sets the buffering policy for the FileHandle. The calling sequence for the Perl function is the same as its C counterpart, including the macros `_IOFBF`, `_IOLBF`, and `_IONBF`, except that the buffer parameter specifies a scalar variable to use as a buffer. WARNING: A variable used as a buffer by `FileHandle::setvbuf` must not be modified in any way until the FileHandle is closed or until `FileHandle::setvbuf` is called again, or memory corruption may result!

See the *perlfunc* manpage for complete descriptions of each of the following supported `FileHandle` methods, which are just front ends for the corresponding built-in functions:

```
close
fileno
getc
gets
eof
clearerr
seek
tell
```

See the *perlvar* manpage for complete descriptions of each of the following supported `FileHandle` methods:

```
autoflush
output_field_separator
output_record_separator
input_record_separator
input_line_number
format_page_number
format_lines_per_page
format_lines_left
format_name
format_top_name
format_line_break_characters
format_formfeed
```

Furthermore, for doing normal I/O you might need these:

$fh->print
: See the `print` entry in the *perlfunc* manpage.

$fh->printf
: See the `printf` entry in the *perlfunc* manpage.

$fh->getline
: This works like <$fh> described in the section on *I/O Operators* in the *perlop* manpage except that it's more readable and can be safely called in an array context but still returns just one line.

$fh->getlines

This works like <$fh> when called in an array context to read all the remaining lines in a file, except that it's more readable. It will also *croak()* if accidentally called in a scalar context.

There are many other functions available since FileHandle is descended from IO::File, IO::Seekable, and IO::Handle. Please see those respective pages for documentation on more functions.

**SEE ALSO**

The **IO** extension, the *perlfunc* manpage, the section on *I/O Operators* in the *perlop* manpage.

**NAME**

FindBin − Locate directory of original perl script

**SYNOPSIS**

```
use FindBin;
use lib "$FindBin::Bin/../lib";

or

use FindBin qw($Bin);
use lib "$Bin/../lib";
```

**DESCRIPTION**

Locates the full path to the script bin directory to allow the use of paths relative to the bin directory.

This allows a user to setup a directory tree for some software with directories <root>/bin and <root>/lib and then the above example will allow the use of modules in the lib directory without knowing where the software tree is installed.

If perl is invoked using the **−e** option or the perl script is read from `STDIN` then FindBin sets both `$Bin` and `$RealBin` to the current directory.

**EXPORTABLE VARIABLES**

```
$Bin         - path to bin directory from where script was invoked
$Script      - basename of script from which perl was invoked
$RealBin     - $Bin with all links resolved
$RealScript  - $Script with all links resolved
```

**KNOWN BUGS**

if perl is invoked as

```
perl filename
```

and *filename* does not have executable rights and a program called *filename* exists in the users `$ENV{PATH}` which satisfies both **−x** and **−T** then FindBin assumes that it was invoked via the `$ENV{PATH}`.

Workaround is to invoke perl as

```
perl ./filename
```

**AUTHORS**

Graham Barr <*bodg@tiuk.ti.com*> Nick Ing-Simmons <*nik@tiuk.ti.com*>

**COPYRIGHT**



**REVISION**

$Revision: 1.4 $

## NAME

GDBM_File − Perl5 access to the gdbm library.

## SYNOPSIS

```
use GDBM_File ;
tie %hash, 'GDBM_File', $filename, &GDBM_WRCREAT, 0640;
# Use the %hash array.
untie %hash ;
```

## DESCRIPTION

**GDBM_File** is a module which allows Perl programs to make use of the facilities provided by the GNU gdbm library. If you intend to use this module you should really have a copy of the gdbm manualpage at hand.

Most of the libgdbm.a functions are available through the GDBM_File interface.

## AVAILABILITY

Gdbm is available from any GNU archive. The master site is `prep.ai.mit.edu`, but your are strongly urged to use one of the many mirrors. You can obtain a list of mirror sites by issuing the command `finger fsf@prep.ai.mit.edu`.

## BUGS

The available functions and the gdbm/perl interface need to be documented.

## SEE ALSO

the *perl(1)* manpage, the *DB_File(3)* manpage.

## NAME

GetOptions – extended processing of command line options

## SYNOPSIS

```
use Getopt::Long;
$result = GetOptions (...option-descriptions...);
```

## DESCRIPTION

The Getopt::Long module implements an extended getopt function called *GetOptions()*. This function adheres to the POSIX syntax for command line options, with GNU extensions. In general, this means that options have long names instead of single letters, and are introduced with a double dash "--". Support for bundling of command line options, as was the case with the more traditional single-letter approach, is provided but not enabled by default. For example, the UNIX "ps" command can be given the command line "option"

```
-vax
```

which means the combination of **–v**, **–a** and **–x**. With the new syntax **--vax** would be a single option, probably indicating a computer architecture.

Command line options can be used to set values. These values can be specified in one of two ways:

```
--size 24
--size=24
```

GetOptions is called with a list of option-descriptions, each of which consists of two elements: the option specifier and the option linkage. The option specifier defines the name of the option and, optionally, the value it can take. The option linkage is usually a reference to a variable that will be set when the option is used. For example, the following call to GetOptions:

```
GetOptions("size=i" => \$offset);
```

will accept a command line option "size" that must have an integer value. With a command line of "— size 24" this will cause the variable `$offset` to get the value 24.

Alternatively, the first argument to GetOptions may be a reference to a HASH describing the linkage for the options, or an object whose class is based on a HASH. The following call is equivalent to the example above:

```
%optctl = ("size" => \$offset);
GetOptions(\%optctl, "size=i");
```

Linkage may be specified using either of the above methods, or both. Linkage specified in the argument list takes precedence over the linkage specified in the HASH.

The command line options are taken from array `@ARGV`. Upon completion of GetOptions, `@ARGV` will contain the rest (i.e. the non-options) of the command line.

Each option specifier designates the name of the option, optionally followed by an argument specifier.

Options that do not take arguments will have no argument specifier. The option variable will be set to 1 if the option is used.

For the other options, the values for argument specifiers are:

! Option does not take an argument and may be negated, i.e. prefixed by "no". E.g. "foo!" will allow **--foo** (with value 1) and **–nofoo** (with value 0). The option variable will be set to 1, or 0 if negated.

\+ Option does not take an argument and will be incremented by 1 every time it appears on the command line. E.g. "more+", when used with **--more --more --more**, will set the option variable to 3 (provided it was 0 or undefined at first).

The **+** specifier is ignored if the option destination is not a SCALAR.

=s Option takes a mandatory string argument. This string will be assigned to the option variable. Note that even if the string argument starts with **–** or **--**, it will not be considered an option on itself.

:s Option takes an optional string argument. This string will be assigned to the option variable. If omitted, it will be assigned "" (an empty string). If the string argument starts with **–** or **--**, it will be considered an option on itself.

=i Option takes a mandatory integer argument. This value will be assigned to the option variable. Note that the value may start with **–** to indicate a negative value.

:i Option takes an optional integer argument. This value will be assigned to the option variable. If omitted, the value 0 will be assigned. Note that the value may start with **–** to indicate a negative value.

=f Option takes a mandatory real number argument. This value will be assigned to the option variable. Note that the value may start with **–** to indicate a negative value.

:f Option takes an optional real number argument. This value will be assigned to the option variable. If omitted, the value 0 will be assigned.

A lone dash **–** is considered an option, the corresponding option name is the empty string.

A double dash on itself **--** signals end of the options list.

**Linkage specification**

The linkage specifier is optional. If no linkage is explicitly specified but a ref HASH is passed, GetOptions will place the value in the HASH. For example:

```
%optctl = ();
GetOptions (\%optctl, "size=i");
```

will perform the equivalent of the assignment

```
$optctl{"size"} = 24;
```

For array options, a reference to an array is used, e.g.:

```
%optctl = ();
GetOptions (\%optctl, "sizes=i@");
```

with command line "–sizes 24 –sizes 48" will perform the equivalent of the assignment

```
$optctl{"sizes"} = [24, 48];
```

For hash options (an option whose argument looks like "name=value"), a reference to a hash is used, e.g.:

```
%optctl = ();
GetOptions (\%optctl, "define=s%");
```

with command line "—define foo=hello --define bar=world" will perform the equivalent of the assignment

```
$optctl{"define"} = {foo=>'hello', bar=>'world')
```

If no linkage is explicitly specified and no ref HASH is passed, GetOptions will put the value in a global variable named after the option, prefixed by "opt_". To yield a usable Perl variable, characters that are not part of the syntax for variables are translated to underscores. For example, "—fpp-struct-return" will set the variable `$opt_fpp_struct_return`. Note that this variable resides in the namespace of the calling program, not necessarily **main**. For example:

```
GetOptions ("size=i", "sizes=i@");
```

with command line "–size 10 –sizes 24 –sizes 48" will perform the equivalent of the assignments

```
$opt_size = 10;
@opt_sizes = (24, 48);
```

A lone dash **–** is considered an option, the corresponding Perl identifier is `$opt_` .

The linkage specifier can be a reference to a scalar, a reference to an array, a reference to a hash or a reference to a subroutine.

Note that, if your code is running under the recommended `use strict 'vars'` pragma, it may be helpful to declare these package variables via `use vars` perhaps something like this:

```
use vars qw/ $opt_size @opt_sizes $opt_bar /;
```

If a REF SCALAR is supplied, the new value is stored in the referenced variable. If the option occurs more than once, the previous value is overwritten.

If a REF ARRAY is supplied, the new value is appended (pushed) to the referenced array.

If a REF HASH is supplied, the option value should look like "key" or "key=value" (if the "=value" is omitted then a value of 1 is implied). In this case, the element of the referenced hash with the key "key" is assigned "value".

If a REF CODE is supplied, the referenced subroutine is called with two arguments: the option name and the option value. The option name is always the true name, not an abbreviation or alias.

### Aliases and abbreviations

The option name may actually be a list of option names, separated by "|"s, e.g. "foo|bar|blech=s". In this example, "foo" is the true name of this option. If no linkage is specified, options "foo", "bar" and "blech" all will set `$opt_foo`. For convenience, the single character "?" is allowed as an alias, e.g. "help|?".

Option names may be abbreviated to uniqueness, depending on configuration option **auto_abbrev**.

### Non-option call-back routine

A special option specifier, <>, can be used to designate a subroutine to handle non-option arguments. GetOptions will immediately call this subroutine for every non-option it encounters in the options list. This subroutine gets the name of the non-option passed. This feature requires configuration option **permute**, see section CONFIGURATION OPTIONS.

See also the examples.

### Option starters

On the command line, options can start with **–** (traditional), **--** (POSIX) and **+** (GNU, now being phased out). The latter is not allowed if the environment variable **POSIXLY_CORRECT** has been defined.

Options that start with "--" may have an argument appended, separated with an "=", e.g. "—foo=bar".

### Return values and Errors

Configuration errors and errors in the option definitions are signalled using `die()` and will terminate the calling program unless the call to `Getopt::Long::GetOptions()` was embedded in `eval { ... }` or `die()` was trapped using `$SIG{__DIE__}`.

A return value of 1 (true) indicates success.

A return status of 0 (false) indicates that the function detected one or more errors during option parsing. These errors are signalled using `warn()` and can be trapped with `$SIG{__WARN__}`.

Errors that can't happen are signalled using `Carp::croak()`.

**COMPATIBILITY**

*Getopt::Long::GetOptions()* is the successor of **newgetopt.pl** that came with Perl 4. It is fully upward compatible. In fact, the Perl 5 version of newgetopt.pl is just a wrapper around the module.

If an "@" sign is appended to the argument specifier, the option is treated as an array. *Value* (s) are not set, but pushed into array `@opt_name`. If explicit linkage is supplied, this must be a reference to an ARRAY.

If an "%" sign is appended to the argument specifier, the option is treated as a hash. *Value* (s) of the form "name=value" are set by setting the element of the hash `%opt_name` with key "name" to "value" (if the "=value" portion is omitted it defaults to 1). If explicit linkage is supplied, this must be a reference to a HASH.

If configuration option **getopt_compat** is set (see section CONFIGURATION OPTIONS), options that start with "+" or "−" may also include their arguments, e.g. "+foo=bar". This is for compatiblity with older implementations of the GNU "getopt" routine.

If the first argument to GetOptions is a string consisting of only non-alphanumeric characters, it is taken to specify the option starter characters. Everything starting with one of these characters from the starter will be considered an option. **Using a starter argument is strongly deprecated.**

For convenience, option specifiers may have a leading **−** or **--**, so it is possible to write:

```
GetOptions qw(-foo=s --bar=i --ar=s);
```

**EXAMPLES**

If the option specifier is "one:i" (i.e. takes an optional integer argument), then the following situations are handled:

```
-one -two            -> $opt_one = '', -two is next option
-one -2              -> $opt_one = -2
```

Also, assume specifiers "foo=s" and "bar:s" :

```
-bar -xxx            -> $opt_bar = '', '-xxx' is next option
-foo -bar            -> $opt_foo = '-bar'
-foo --              -> $opt_foo = '--'
```

In GNU or POSIX format, option names and values can be combined:

```
+foo=blech           -> $opt_foo = 'blech'
--bar=               -> $opt_bar = ''
--bar=--             -> $opt_bar = '--'
```

Example of using variable references:

```
$ret = GetOptions ('foo=s', \$foo, 'bar=i', 'ar=s', \@ar);
```

With command line options "−foo blech −bar 24 −ar xx −ar yy" this will result in:

```
$foo = 'blech'
$opt_bar = 24
@ar = ('xx','yy')
```

Example of using the <> option specifier:

```
@ARGV = qw(-foo 1 bar -foo 2 blech);
GetOptions("foo=i", \$myfoo, "<>", \&mysub);
```

Results:

```
mysub("bar") will be called (with $myfoo being 1)
mysub("blech") will be called (with $myfoo being 2)
```

Compare this with:

```
@ARGV = qw(-foo 1 bar -foo 2 blech);
GetOptions("foo=i", \$myfoo);
```

This will leave the non-options in `@ARGV`:

```
$myfoo -> 2
@ARGV -> qw(bar blech)
```

## CONFIGURATION OPTIONS

**GetOptions** can be configured by calling subroutine **Getopt::Long::Configure**. This subroutine takes a list of quoted strings, each specifying a configuration option to be set, e.g. **ignore_case**. Options can be reset by prefixing with **no_**, e.g. **no_ignore_case**. Case does not matter. Multiple calls to **config** are possible.

Previous versions of Getopt::Long used variables for the purpose of configuring. Although manipulating these variables still work, it is strongly encouraged to use the new **config** routine. Besides, it is much easier.

The following options are available:

default
: This option causes all configuration options to be reset to their default values.

auto_abbrev
: Allow option names to be abbreviated to uniqueness. Default is set unless environment variable POSIXLY_CORRECT has been set, in which case **auto_abbrev** is reset.

getopt_compat
: Allow '+' to start options. Default is set unless environment variable POSIXLY_CORRECT has been set, in which case **getopt_compat** is reset.

require_order
: Whether non-options are allowed to be mixed with options. Default is set unless environment variable POSIXLY_CORRECT has been set, in which case b<require_order> is reset.

See also **permute**, which is the opposite of **require_order**.

permute
: Whether non-options are allowed to be mixed with options. Default is set unless environment variable POSIXLY_CORRECT has been set, in which case **permute** is reset. Note that **permute** is the opposite of **require_order**.

If **permute** is set, this means that

```
-foo arg1 -bar arg2 arg3
```

is equivalent to

```
-foo -bar arg1 arg2 arg3
```

If a non-option call-back routine is specified, `@ARGV` will always be empty upon succesful return of GetOptions since all options have been processed, except when **--** is used:

```
-foo arg1 -bar arg2 -- arg3
```

will call the call-back routine for arg1 and arg2, and terminate leaving arg2 in `@ARGV`.

If **require_order** is set, options processing terminates when the first non-option is encountered.

```
-foo arg1 -bar arg2 arg3
```

is equivalent to

```
-foo -- arg1 -bar arg2 arg3
```

bundling (default: reset)
Setting this variable to a non-zero value will allow single-character options to be bundled. To distinguish bundles from long option names, long options must be introduced with **--** and single-character options (and bundles) with **–**. For example,

```
ps -vax --vax
```

would be equivalent to

```
ps -v -a -x --vax
```

provided "vax", "v", "a" and "x" have been defined to be valid options.

Bundled options can also include a value in the bundle; for strings this value is the rest of the bundle, but integer and floating values may be combined in the bundle, e.g.

```
scale -h24w80
```

is equivalent to

```
scale -h 24 -w 80
```

Note: resetting **bundling** also resets **bundling_override**.

bundling_override (default: reset)
If **bundling_override** is set, bundling is enabled as with **bundling** but now long option names override option bundles. In the above example, **–vax** would be interpreted as the option "vax", not the bundle "v", "a", "x".

Note: resetting **bundling_override** also resets **bundling**.

**Note:** Using option bundling can easily lead to unexpected results, especially when mixing long options and bundles. Caveat emptor.

ignore_case (default: set)
If set, case is ignored when matching options.

Note: resetting **ignore_case** also resets **ignore_case_always**.

ignore_case_always (default: reset)
When bundling is in effect, case is ignored on single-character options also.

Note: resetting **ignore_case_always** also resets **ignore_case**.

pass_through (default: reset)
Unknown options are passed through in `@ARGV` instead of being flagged as errors. This makes it possible to write wrapper scripts that process only part of the user supplied options, and passes the remaining options to some other program.

This can be very confusing, especially when **permute** is also set.

prefix
The string that starts options. See also **prefix_pattern**.

prefix_pattern
A Perl pattern that identifies the strings that introduce options. Default is `(--|-|\+)` unless environment variable POSIXLY_CORRECT has been set, in which case it is `(--|-)`.

debug (default: reset)
Enable copious debugging output.

**OTHER USEFUL VARIABLES**

$Getopt::Long::VERSION
The version number of this Getopt::Long implementation in the format `major.minor`. This can be used to have Exporter check the version, e.g.

```
use Getopt::Long 3.00;
```

You can inspect `$Getopt::Long::major_version` and `$Getopt::Long::minor_version` for the individual components.

$Getopt::Long::error
Internal error flag. May be incremented from a call-back routine to cause options parsing to fail.

**AUTHOR**

Johan Vromans <jvromans@squirrel.nl>

**COPYRIGHT AND DISCLAIMER**

**NAME**

getopt − Process single-character switches with switch clustering

getopts − Process single-character switches with switch clustering

**SYNOPSIS**

```
use Getopt::Std;

getopt('oDI');    # -o, -D & -I take arg.  Sets opt_* as a side effect.
getopt('oDI', \%opts);    # -o, -D & -I take arg.  Values in %opts
getopts('oif:');  # -o & -i are boolean flags, -f takes an argument
                  # Sets opt_* as a side effect.
getopts('oif:', \%opts);  # options as above. Values in %opts
```

**DESCRIPTION**

The *getopt()* functions processes single-character switches with switch clustering. Pass one argument which is a string containing all switches that take an argument. For each switch found, sets `$opt_x` (where x is the switch name) to the value of the argument, or 1 if no argument. Switches which take an argument don't care whether there is a space between the switch and the argument.

Note that, if your code is running under the recommended `use strict 'vars'` pragma, it may be helpful to declare these package variables via `use vars` perhaps something like this:

```
use vars qw/ $opt_foo $opt_bar /;
```

For those of you who don't like additional variables being created, *getopt()* and *getopts()* will also accept a hash reference as an optional second argument. Hash keys will be x (where x is the switch name) with key values the value of the argument or 1 if no argument is specified.

## NAME

I18N::Collate − compare 8-bit scalar data according to the current locale

```
***

WARNING: starting from the Perl version 5.003_06
the I18N::Collate interface for comparing 8−bit scalar data
according to the current locale

      HAS BEEN DEPRECATED

That is, please do not use it anymore for any new applications
and please migrate the old applications away from it because its
functionality was integrated into the Perl core language in the
release 5.003_06.

See the perllocale manual page for further information.

***
```

## SYNOPSIS

```
use I18N::Collate;
setlocale(LC_COLLATE, 'locale−of−your−choice');
$s1 = new I18N::Collate "scalar_data_1";
$s2 = new I18N::Collate "scalar_data_2";
```

## DESCRIPTION

This module provides you with objects that will collate according to your national character set, provided that the POSIX *setlocale()* function is supported on your system.

You can compare `$s1` and `$s2` above with

```
$s1 le $s2
```

to extract the data itself, you'll need a dereference: $$s1

This module uses *POSIX::setlocale()*. The basic collation conversion is done by *strxfrm()* which terminates at NUL characters being a decent C routine. *collate_xfrm()* handles embedded NUL characters gracefully.

The available locales depend on your operating system; try whether `locale −a` shows them or man pages for "locale" or "nlsinfo" or the direct approach `ls /usr/lib/nls/loc` or `ls /usr/lib/nls` or `ls /usr/lib/locale`. Not all the locales that your vendor supports are necessarily installed: please consult your operating system's documentation and possibly your local system administration. The locale names are probably something like `xx_XX.(ISO)?8859−N` or `xx_XX.(ISO)?8859N`, for example `fr_CH.ISO8859−1` is the Swiss (CH) variant of French (fr), ISO Latin (8859) 1 (−1) which is the Western European character set.

**NAME**

IO − load various IO modules

**SYNOPSIS**

```
use IO;
```

**DESCRIPTION**

`IO` provides a simple mechanism to load some of the IO modules at one go. Currently this includes:

```
IO::Handle
IO::Seekable
IO::File
IO::Pipe
IO::Socket
```

For more information on any of these modules, please see its respective documentation.

**NAME**

IO::File − supply object methods for filehandles

**SYNOPSIS**

```
use IO::File;

$fh = new IO::File;
if ($fh->open("< file")) {
    print <$fh>;
    $fh->close;
}

$fh = new IO::File "> file";
if (defined $fh) {
    print $fh "bar\n";
    $fh->close;
}

$fh = new IO::File "file", "r";
if (defined $fh) {
    print <$fh>;
    undef $fh;       # automatically closes the file
}

$fh = new IO::File "file", O_WRONLY|O_APPEND;
if (defined $fh) {
    print $fh "corge\n";

    $pos = $fh->getpos;
    $fh->setpos($pos);

    undef $fh;       # automatically closes the file
}

autoflush STDOUT 1;
```

**DESCRIPTION**

`IO::File` inherits from `IO::Handle` and `IO::Seekable`. It extends these classes with methods that are specific to file handles.

**CONSTRUCTOR**

new ([ ARGS ] )

Creates a `IO::File`. If it receives any parameters, they are passed to the method `open`; if the open fails, the object is destroyed. Otherwise, it is returned to the caller.

new_tmpfile

Creates an `IO::File` opened for read/write on a newly created temporary file. On systems where this is possible, the temporary file is anonymous (i.e. it is unlinked after creation, but held open). If the temporary file cannot be created or opened, the `IO::File` object is destroyed. Otherwise, it is returned to the caller.

**METHODS**

open( FILENAME [,MODE [,PERMS]] )

`open` accepts one, two or three parameters. With one parameter, it is just a front end for the built-in `open` function. With two parameters, the first parameter is a filename that may include whitespace or other special characters, and the second parameter is the open mode, optionally followed by a file permission value.

If `IO::File::open` receives a Perl mode string (">", "+<", etc.) or a POSIX *fopen()* mode string ("w", "r+", etc.), it uses the basic Perl `open` operator.

If `IO::File::open` is given a numeric mode, it passes that mode and the optional permissions value to the Perl `sysopen` operator. For convenience, `IO::File::import` tries to import the O_XXX constants from the Fcntl module. If dynamic loading is not available, this may fail, but the rest of IO::File will still work.

**SEE ALSO**

the *perlfunc* manpage, the section on *I/O Operators* in the *perlop* manpage, the *IO::Handle* manpage the *IO::Seekable* manpage

**HISTORY**

Derived from FileHandle.pm by Graham Barr <*bodg@tiuk.ti.com*>.

**NAME**

IO::Handle – supply object methods for I/O handles

**SYNOPSIS**

```
use IO::Handle;

$fh = new IO::Handle;
if ($fh->fdopen(fileno(STDIN),"r")) {
    print $fh->getline;
    $fh->close;
}

$fh = new IO::Handle;
if ($fh->fdopen(fileno(STDOUT),"w")) {
    $fh->print("Some text\n");
}

use IO::Handle '_IOLBF';
$fh->setvbuf($buffer_var, _IOLBF, 1024);

undef $fh;       # automatically closes the file if it's open

autoflush STDOUT 1;
```

**DESCRIPTION**

`IO::Handle` is the base class for all other IO handle classes. It is not intended that objects of `IO::Handle` would be created directly, but instead `IO::Handle` is inherited from by several other classes in the IO hierarchy.

If you are reading this documentation, looking for a replacement for the `FileHandle` package, then I suggest you read the documentation for `IO::File`

A `IO::Handle` object is a reference to a symbol (see the `Symbol` package)

**CONSTRUCTOR**

new ()

Creates a new `IO::Handle` object.

new_from_fd ( FD, MODE )

Creates a `IO::Handle` like `new` does. It requires two parameters, which are passed to the method `fdopen`; if the fdopen fails, the object is destroyed. Otherwise, it is returned to the caller.

**METHODS**

See the *perlfunc* manpage for complete descriptions of each of the following supported `IO::Handle` methods, which are just front ends for the corresponding built-in functions:

```
close
fileno
getc
eof
read
truncate
stat
print
printf
sysread
syswrite
```

See the *perlvar* manpage for complete descriptions of each of the following supported `IO::Handle` methods:

```
autoflush
output_field_separator
output_record_separator
input_record_separator
input_line_number
format_page_number
format_lines_per_page
format_lines_left
format_name
format_top_name
format_line_break_characters
format_formfeed
format_write
```

Furthermore, for doing normal I/O you might need these:

$fh->fdopen ( FD, MODE )
: `fdopen` is like an ordinary `open` except that its first parameter is not a filename but rather a file handle name, a IO::Handle object, or a file descriptor number.

$fh->opened
: Returns true if the object is currently a valid file descriptor.

$fh->getline
: This works like <$fh> described in the section on *I/O Operators* in the *perlop* manpage except that it's more readable and can be safely called in an array context but still returns just one line.

$fh->getlines
: This works like <$fh> when called in an array context to read all the remaining lines in a file, except that it's more readable. It will also *croak()* if accidentally called in a scalar context.

$fh->ungetc ( ORD )
: Pushes a character with the given ordinal value back onto the given handle's input stream.

$fh->write ( BUF, LEN [, OFFSET }\] )
: This `write` is like `write` found in C, that is it is the opposite of read. The wrapper for the perl `write` function is called `format_write`.

$fh->flush
: Flush the given handle's buffer.

$fh->error
: Returns a true value if the given handle has experienced any errors since it was opened or since the last call to `clearerr`.

$fh->clearerr
: Clear the given handle's error indicator.

If the C functions *setbuf()* and/or *setvbuf()* are available, then `IO::Handle::setbuf` and `IO::Handle::setvbuf` set the buffering policy for an IO::Handle. The calling sequences for the Perl functions are the same as their C counterparts— including the constants `_IOFBF`, `_IOLBF`, and `_IONBF` for *setvbuf()*--except that the buffer parameter specifies a scalar variable to use as a buffer. WARNING: A variable used as a buffer by `setbuf` or `setvbuf` must not be modified in any way until the IO::Handle is closed or `setbuf` or `setvbuf` is called again, or memory corruption may result! Note that you need to import the constants `_IOFBF`, `_IOLBF`, and `_IONBF` explicitly.

Lastly, there is a special method for working under **–T** and setuid/gid scripts:

$fh->untaint

Marks the object as taint-clean, and as such data read from it will also be considered taint-clean. Note that this is a very trusting action to take, and appropriate consideration for the data source and potential vulnerability should be kept in mind.

**NOTE**

A `IO::Handle` object is a GLOB reference. Some modules that inherit from `IO::Handle` may want to keep object related variables in the hash table part of the GLOB. In an attempt to prevent modules trampling on each other I propose the that any such module should prefix its variables with its own name separated by _'s. For example the IO::Socket module keeps a `timeout` variable in 'io_socket_timeout'.

**SEE ALSO**

the *perlfunc* manpage, the section on *I/O Operators* in the *perlop* manpage, the *IO::File* manpage

**BUGS**

Due to backwards compatibility, all filehandles resemble objects of class `IO::Handle`, or actually classes derived from that class. They actually aren't. Which means you can't derive your own class from `IO::Handle` and inherit those methods.

**HISTORY**

Derived from FileHandle.pm by Graham Barr <*bodg@tiuk.ti.com*>

## NAME

IO::pipe − supply object methods for pipes

## SYNOPSIS

```
use IO::Pipe;

$pipe = new IO::Pipe;

if($pid = fork()) { # Parent
    $pipe->reader();

    while(<$pipe> {
        ....
    }

}
elsif(defined $pid) { # Child
    $pipe->writer();

    print $pipe ....
}

or

$pipe = new IO::Pipe;

$pipe->reader(qw(ls -l));

while(<$pipe>) {
    ....
}
```

## DESCRIPTION

`IO::Pipe` provides an interface to createing pipes between processes.

## CONSTRCUTOR

new ( [READER, WRITER] )

Creates a `IO::Pipe`, which is a reference to a newly created symbol (see the `Symbol` package). `IO::Pipe::new` optionally takes two arguments, which should be objects blessed into `IO::Handle`, or a subclass thereof. These two objects will be used for the system call to `pipe`. If no arguments are given then method `handles` is called on the new `IO::Pipe` object.

These two handles are held in the array part of the GLOB until either `reader` or `writer` is called.

## METHODS

reader ([ARGS])

The object is re-blessed into a sub-class of `IO::Handle`, and becomes a handle at the reading end of the pipe. If `ARGS` are given then `fork` is called and `ARGS` are passed to exec.

writer ([ARGS])

The object is re-blessed into a sub-class of `IO::Handle`, and becomes a handle at the writing end of the pipe. If `ARGS` are given then `fork` is called and `ARGS` are passed to exec.

handles ()

This method is called during construction by `IO::Pipe::new` on the newly created `IO::Pipe` object. It returns an array of two objects blessed into `IO::Pipe::End`, or a subclass thereof.

**SEE ALSO**

the *IO::Handle* manpage

**AUTHOR**

Graham Barr <bodg@tiuk.ti.com>

**COPYRIGHT**

**NAME**

IO::Seekable − supply seek based methods for I/O objects

**SYNOPSIS**

```
use IO::Seekable;
package IO::Something;
@ISA = qw(IO::Seekable);
```

**DESCRIPTION**

`IO::Seekable` does not have a constuctor of its own as is intended to be inherited by other `IO::Handle` based objects. It provides methods which allow seeking of the file descriptors.

If the C functions *fgetpos()* and *fsetpos()* are available, then `IO::File::getpos` returns an opaque value that represents the current position of the IO::File, and `IO::File::setpos` uses that value to return to a previously visited position.

See the *perlfunc* manpage for complete descriptions of each of the following supported `IO::Seekable` methods, which are just front ends for the corresponding built-in functions:

```
seek
tell
```

**SEE ALSO**

the *perlfunc* manpage, the section on *I/O Operators* in the *perlop* manpage, the *IO::Handle* manpage the *IO::File* manpage

**HISTORY**

Derived from FileHandle.pm by Graham Barr <bodg@tiuk.ti.com>

**NAME**

IO::Select – OO interface to the select system call

**SYNOPSIS**

```
use IO::Select;

$s = IO::Select->new();

$s->add(\*STDIN);
$s->add($some_handle);

@ready = $s->can_read($timeout);

@ready = IO::Select->new(@handles)->read(0);
```

**DESCRIPTION**

The `IO::Select` package implements an object approach to the system `select` function call. It allows the user to see what IO handles, see the *IO::Handle* manpage, are ready for reading, writing or have an error condition pending.

**CONSTRUCTOR**

new ( [ HANDLES ] )

The constructor creates a new object and optionally initialises it with a set of handles.

**METHODS**

add ( HANDLES )

Add the list of handles to the `IO::Select` object. It is these values that will be returned when an event occurs. `IO::Select` keeps these values in a cache which is indexed by the `fileno` of the handle, so if more than one handle with the same `fileno` is specified then only the last one is cached.

Each handle can be an `IO::Handle` object, an integer or an array reference where the first element is a `IO::Handle` or an integer.

remove ( HANDLES )

Remove all the given handles from the object. This method also works by the `fileno` of the handles. So the exact handles that were added need not be passed, just handles that have an equivalent `fileno`

exists ( HANDLE )

Returns a true value (actually the handle itself) if it is present. Returns undef otherwise.

handles

Return an array of all registered handles.

can_read ( [ TIMEOUT ] )

Return an array of handles that are ready for reading. `TIMEOUT` is the maximum amount of time to wait before returning an empty list. If `TIMEOUT` is not given and any handles are registered then the call will block.

can_write ( [ TIMEOUT ] )

Same as `can_read` except check for handles that can be written to.

has_error ( [ TIMEOUT ] )

Same as `can_read` except check for handles that have an error condition, for example EOF.

count ()

Returns the number of handles that the object will check for when one of the `can_` methods is called or the object is passed to the `select` static method.

bits()

Return the bit string suitable as argument to the core *select()* call.

bits()
Return the bit string suitable as argument to the core *select()* call.

select ( READ, WRITE, ERROR [, TIMEOUT ] )
`select` is a static method, that is you call it with the package name like `new`. `READ`, `WRITE` and `ERROR` are either `undef` or `IO::Select` objects. `TIMEOUT` is optional and has the same effect as for the core select call.

The result will be an array of 3 elements, each a reference to an array which will hold the handles that are ready for reading, writing and have error conditions respectively. Upon error an empty array is returned.

**EXAMPLE**

Here is a short example which shows how `IO::Select` could be used to write a server which communicates with several sockets while also listening for more connections on a listen socket

```
use IO::Select;
use IO::Socket;

$lsn = new IO::Socket::INET(Listen => 1, LocalPort => 8080);
$sel = new IO::Select( $lsn );

while(@ready = $sel->can_read) {
    foreach $fh (@ready) {
        if($fh == $lsn) {
            # Create a new socket
            $new = $lsn->accept;
            $sel->add($new);
        }
        else {
            # Process socket

            # Maybe we have finished with the socket
            $sel->remove($fh);
            $fh->close;
        }
    }
}
```

**AUTHOR**

Graham Barr <*Graham.Barr@tiuk.ti.com*>

**COPYRIGHT**

**NAME**

IO::Socket – Object interface to socket communications

**SYNOPSIS**

```
use IO::Socket;
```

**DESCRIPTION**

`IO::Socket` provides an object interface to creating and using sockets. It is built upon the the *IO::Handle* manpage interface and inherits all the methods defined by the *IO::Handle* manpage.

`IO::Socket` only defines methods for those operations which are common to all types of socket. Operations which are specified to a socket in a particular domain have methods defined in sub classes of `IO::Socket`

`IO::Socket` will export all functions (and constants) defined by the *Socket* manpage.

**CONSTRUCTOR**

new ( [ARGS] )

Creates an `IO::Socket`, which is a reference to a newly created symbol (see the `Symbol` package). `new` optionally takes arguments, these arguments are in key-value pairs. `new` only looks for one key `Domain` which tells new which domain the socket will be in. All other arguments will be passed to the configuration method of the package for that domain, See below.

`IO::Sockets` will be in autoflush mode after creation. Note that versions of IO::Socket prior to 1.1603 (as shipped with Perl 5.004_04) did not do this. So if you need backward compatibility, you should set autoflush explicitly.

**METHODS**

See the *perlfunc* manpage for complete descriptions of each of the following supported `IO::Socket` methods, which are just front ends for the corresponding built-in functions:

```
socket
socketpair
bind
listen
accept
send
recv
peername (getpeername)
sockname (getsockname)
```

Some methods take slightly different arguments to those defined in the *perlfunc* manpage in attempt to make the interface more flexible. These are

accept([PKG])

perform the system call `accept` on the socket and return a new object. The new object will be created in the same class as the listen socket, unless `PKG` is specified. This object can be used to communicate with the client that was trying to connect. In a scalar context the new socket is returned, or undef upon failure. In an array context a two-element array is returned containing the new socket and the peer address, the list will be empty upon failure.

Additional methods that are provided are

timeout([VAL])

Set or get the timeout value associated with this socket. If called without any arguments then the current setting is returned. If called with an argument the current setting is changed and the previous value returned.

sockopt(OPT [, VAL])
: Unified method to both set and get options in the SOL_SOCKET level. If called with one argument then getsockopt is called, otherwise setsockopt is called.

sockdomain
: Returns the numerical number for the socket domain type. For example, for a AF_INET socket the value of &AF_INET will be returned.

socktype
: Returns the numerical number for the socket type. For example, for a SOCK_STREAM socket the value of &SOCK_STREAM will be returned.

protocol
: Returns the numerical number for the protocol being used on the socket, if known. If the protocol is unknown, as with an AF_UNIX socket, zero is returned.

## SUB–CLASSES

### IO::Socket::INET

`IO::Socket::INET` provides a constructor to create an AF_INET domain socket and some related methods. The constructor can take the following options

```
PeerAddr    Remote host address          <hostname>[:<port>]
PeerPort    Remote port or service       <service>[(<no>)] | <no>
LocalAddr   Local host bind address      hostname[:port]
LocalPort   Local host bind port         <service>[(<no>)] | <no>
Proto       Protocol name (or number)    "tcp" | "udp" | ...
Type        Socket type                  SOCK_STREAM | SOCK_DGRAM | ...
Listen      Queue size for listen
Reuse       Set SO_REUSEADDR before binding
Timeout     Timeout value for various operations
```

If `Listen` is defined then a listen socket is created, else if the socket type, which is derived from the protocol, is SOCK_STREAM then *connect()* is called.

The `PeerAddr` can be a hostname or the IP–address on the "xx.xx.xx.xx" form. The `PeerPort` can be a number or a symbolic service name. The service name might be followed by a number in parenthesis which is used if the service is not known by the system. The `PeerPort` specification can also be embedded in the `PeerAddr` by preceding it with a ":".

If `Proto` is not given and you specify a symbolic `PeerPort` port, then the constructor will try to derive `Proto` from the service name. As a last resort `Proto` "tcp" is assumed. The `Type` parameter will be deduced from `Proto` if not specified.

If the constructor is only passed a single argument, it is assumed to be a `PeerAddr` specification.

Examples:

```
$sock = IO::Socket::INET->new(PeerAddr => 'www.perl.org',
                              PeerPort => 'http(80)',
                              Proto    => 'tcp');

$sock = IO::Socket::INET->new(PeerAddr => 'localhost:smtp(25)');

$sock = IO::Socket::INET->new(Listen    => 5,
                              LocalAddr => 'localhost',
                              LocalPort => 9000,
                              Proto     => 'tcp');

$sock = IO::Socket::INET->new('127.0.0.1:25');
```

#### METHODS

sockaddr ()
: Return the address part of the sockaddr structure for the socket

sockport ()
: Return the port number that the socket is using on the local host

sockhost ()
: Return the address part of the sockaddr structure for the socket in a text form xx.xx.xx.xx

peeraddr ()
: Return the address part of the sockaddr structure for the socket on the peer host

peerport ()
: Return the port number for the socket on the peer host.

peerhost ()
: Return the address part of the sockaddr structure for the socket on the peer host in a text form xx.xx.xx.xx

### IO::Socket::UNIX

`IO::Socket::UNIX` provides a constructor to create an AF_UNIX domain socket and some related methods. The constructor can take the following options

```
Type        Type of socket (eg SOCK_STREAM or SOCK_DGRAM)
Local       Path to local fifo
Peer        Path to peer fifo
Listen      Create a listen socket
```

#### METHODS

hostpath()
: Returns the pathname to the fifo at the local end

peerpath()
: Returns the pathanme to the fifo at the peer end

## SEE ALSO

the *Socket* manpage, the *IO::Handle* manpage

## AUTHOR

Graham Barr <*Graham.Barr@tiuk.ti.com*>

## COPYRIGHT

**NAME**

IPC::Msg – SysV Msg IPC object class

**SYNOPSIS**

```
use IPC::SysV qw(IPC_PRIVATE S_IRWXU S_IRWXG S_IRWXO);
use IPC::Msg;

$msg = new IPC::Msg(IPC_PRIVATE, S_IRWXU | S_IRWXG | S_IRWXO);

$msg->snd(pack("L a*",$msgtype,$msg));

$msg->rcv($buf,256);

$ds = $msg->stat;

$msg->remove;
```

**DESCRIPTION**

**METHODS**

new ( KEY , FLAGS )
: Creates a new message queue associated with `KEY`. A new queue is created if

- `KEY` is equal to `IPC_PRIVATE`
- `KEY` does not already have a message queue associated with it, and *FLAGS* & IPC_CREAT is true.

  On creation of a new message queue `FLAGS` is used to set the permissions.

id
: Returns the system message queue identifier.

rcv ( BUF, LEN [, TYPE [, FLAGS ]] )
: Read a message from the queue. Returns the type of the message read. See the *msgrcv* manpage

remove
: Remove and destroy the message queue from the system.

set ( STAT )

set ( NAME => VALUE [, NAME => VALUE ...] )
: `set` will set the following values of the `stat` structure associated with the message queue.

```
uid
gid
mode (oly the permission bits)
qbytes
```

`set` accepts either a stat object, as returned by the `stat` method, or a list of *name–value* pairs.

snd ( TYPE, MSG [, FLAGS ] )
: Place a message on the queue with the data from `MSG` and with type `TYPE`. See the *msgsnd* manpage.

stat
: Returns an object of type `IPC::Msg::stat` which is a sub-class of `Class::Struct`. It provides the following fields. For a description of these fields see you system documentation.

```
uid
gid
cuid
cgid
mode
qnum
qbytes
lspid
lrpid
stime
rtime
ctime
```

**SEE ALSO**

the *IPC::SysV* manpage the *Class::Struct* manpage

**AUTHOR**

Graham Barr <gbarr@pobox.com>

**COPYRIGHT**

## NAME

IPC::Open2, open2 – open a process for both reading and writing

## SYNOPSIS

```
use IPC::Open2;
$pid = open2(\*RDR, \*WTR, 'some cmd and args');
  # or
$pid = open2(\*RDR, \*WTR, 'some', 'cmd', 'and', 'args');
```

## DESCRIPTION

The *open2()* function spawns the given `$cmd` and connects `$rdr` for reading and `$wtr` for writing. It's what you think should work when you try

```
open(HANDLE, "|cmd args|");
```

The write filehandle will have autoflush turned on.

If `$rdr` is a string (that is, a bareword filehandle rather than a glob or a reference) and it begins with ">&", then the child will send output directly to that file handle. If `$wtr` is a string that begins with "<&", then WTR will be closed in the parent, and the child will read from it directly. In both cases, there will be a *dup* (2) instead of a *pipe* (2) made.

*open2()* returns the process ID of the child process. It doesn't return on failure: it just raises an exception matching `/^open2:/`.

## WARNING

It will not create these file handles for you. You have to do this yourself. So don't pass it empty variables expecting them to get filled in for you.

Additionally, this is very dangerous as you may block forever. It assumes it's going to talk to something like **bc**, both writing to it and reading from it. This is presumably safe because you "know" that commands like **bc** will read a line at a time and output a line at a time. Programs like **sort** that read their entire input stream first, however, are quite apt to cause deadlock.

The big problem with this approach is that if you don't have control over source code being run in the child process, you can't control what it does with pipe buffering. Thus you can't just open a pipe to `cat -v` and continually read and write a line from it.

## SEE ALSO

See the *IPC::Open3* manpage for an alternative that handles STDERR as well. This function is really just a wrapper around *open3()*.

**NAME**

IPC::Open3, open3 − open a process for reading, writing, and error handling

**SYNOPSIS**

```
$pid = open3(\*WTRFH, \*RDRFH, \*ERRFH,
                'some cmd and args', 'optarg', ...);
```

**DESCRIPTION**

Extremely similar to *open2()*, *open3()* spawns the given `$cmd` and connects RDRFH for reading, WTRFH for writing, and ERRFH for errors. If ERRFH is '', or the same as RDRFH, then STDOUT and STDERR of the child are on the same file handle. The WTRFH will have autoflush turned on.

If WTRFH begins with "<&", then WTRFH will be closed in the parent, and the child will read from it directly. If RDRFH or ERRFH begins with ">&", then the child will send output directly to that file handle. In both cases, there will be a *dup* (2) instead of a *pipe* (2) made.

If you try to read from the child's stdout writer and their stderr writer, you'll have problems with blocking, which means you'll want to use *select()*, which means you'll have to use *sysread()* instead of normal stuff.

*open3()* returns the process ID of the child process. It doesn't return on failure: it just raises an exception matching `/^open3:/`.

**WARNING**

It will not create these file handles for you. You have to do this yourself. So don't pass it empty variables expecting them to get filled in for you.

Additionally, this is very dangerous as you may block forever. It assumes it's going to talk to something like **bc**, both writing to it and reading from it. This is presumably safe because you "know" that commands like **bc** will read a line at a time and output a line at a time. Programs like **sort** that read their entire input stream first, however, are quite apt to cause deadlock.

The big problem with this approach is that if you don't have control over source code being run in the child process, you can't control what it does with pipe buffering. Thus you can't just open a pipe to `cat -v` and continually read and write a line from it.

**NAME**

IPC::Semaphore − SysV Semaphore IPC object class

**SYNOPSIS**

```
use IPC::SysV qw(IPC_PRIVATE S_IRWXU IPC_CREAT);
use IPC::Semaphore;

$sem = new IPC::Semaphore(IPC_PRIVATE, 10, S_IRWXU | IPC_CREAT);

$sem->setall( (0) x 10);

@sem = $sem->getall;

$ncnt = $sem->getncnt;

$zcnt = $sem->getzcnt;

$ds = $sem->stat;

$sem->remove;
```

**DESCRIPTION**

**METHODS**

new ( KEY , NSEMS , FLAGS )
: Create a new semaphore set associated with `KEY`. `NSEMS` is the number of semaphores in the set. A new set is created if

- `KEY` is equal to `IPC_PRIVATE`
- `KEY` does not already have a semaphore identifier associated with it, and *FLAGS* & IPC_CREAT is true.

  On creation of a new semaphore set `FLAGS` is used to set the permissions.

getall
: Returns the values of the semaphore set as an array.

getncnt ( SEM )
: Returns the number of processed waiting for the semaphore `SEM` to become greater than it's current value

getpid ( SEM )
: Returns the process id of the last process that performed an operation on the semaphore `SEM`.

getval ( SEM )
: Returns the current value of the semaphore `SEM`.

getzcnt ( SEM )
: Returns the number of processed waiting for the semaphore `SEM` to become zero.

id
: Returns the system identifier for the semaphore set.

op ( OPLIST )
: `OPLIST` is a list of operations to pass to `semop`. `OPLIST` is a concatenation of smaller lists, each which has three values. The first is the semaphore number, the second is the operation and the last is a flags value. See the *semop* manpage for more details. For example

```
$sem->op(
    0, -1, IPC_NOWAIT,
    1,  1, IPC_NOWAIT
);
```

remove
: Remove and destroy the semaphore set from the system.

set ( STAT )

set ( NAME => VALUE [, NAME => VALUE ...] )
: `set` will set the following values of the `stat` structure associated with the semaphore set.

```
uid
gid
mode (oly the permission bits)
```

`set` accepts either a stat object, as returned by the `stat` method, or a list of *name–value* pairs.

setall ( VALUES )
: Sets all values in the semaphore set to those given on the `VALUES` list. `VALUES` must contain the correct number of values.

setval ( N , VALUE )
: Set the `N`th value in the semaphore set to `VALUE`

stat
: Returns an object of type `IPC::Semaphore::stat` which is a sub-class of `Class::Struct`. It provides the following fields. For a description of these fields see you system documentation.

```
uid
gid
cuid
cgid
mode
ctime
otime
nsems
```

**SEE ALSO**

the *IPC::SysV* manpage the *Class::Struct* manpage the *semget* manpage the *semctl* manpage the *semop* manpage

**AUTHOR**

Graham Barr <gbarr@pobox.com>

**COPYRIGHT**

## NAME

IPC::SysV – SysV IPC constants

## SYNOPSIS

```
use IPC::SysV qw(IPC_STAT IPC_PRIVATE);
```

## DESCRIPTION

`IPC::SysV` defines and conditionally exports all the constants defined in your system include files which are needed by the SysV IPC calls.

ftok( PATH, ID )
Return a key based on PATH and ID, which can be used as a key for `msgget`, `semget` and `shmget`. See the *ftok* manpage

## SEE ALSO

the *IPC::Msg* manpage, the *IPC::Semaphore* manpage, the *ftok* manpage

## AUTHORS

Graham Barr <gbarr@pobox.com> Jarkko Hietaniemi <jhi@iki.fi>

## COPYRIGHT

**NAME**

Math::BigFloat − Arbitrary length float math package

**SYNOPSIS**

```
use Math::BigFloat;
$f = Math::BigFloat->new($string);

$f->fadd(NSTR) return NSTR            addition
$f->fsub(NSTR) return NSTR            subtraction
$f->fmul(NSTR) return NSTR            multiplication
$f->fdiv(NSTR[,SCALE]) returns NSTR   division to SCALE places
$f->fneg() return NSTR                negation
$f->fabs() return NSTR                absolute value
$f->fcmp(NSTR) return CODE            compare undef,<0,=0,>0
$f->fround(SCALE) return NSTR         round to SCALE digits
$f->ffround(SCALE) return NSTR        round at SCALEth place
$f->fnorm() return (NSTR)             normalize
$f->fsqrt([SCALE]) return NSTR        sqrt to SCALE places
```

**DESCRIPTION**

All basic math operations are overloaded if you declare your big floats as

```
$float = new Math::BigFloat "2.123123123123123123123123123123123";
```

number format
: canonical strings have the form /[+−]\d+E[+−]\d+/ . Input values can have inbedded whitespace.

Error returns 'NaN'
: An input parameter was "Not a Number" or divide by zero or sqrt of negative number.

Division is computed to
: `max($div_scale,length(dividend)+length(divisor))` digits by default. Also used for default sqrt scale.

**BUGS**

The current version of this module is a preliminary version of the real thing that is currently (as of perl5.002) under development.

**AUTHOR**

Mark Biggar

## NAME

Math::BigInt − Arbitrary size integer math package

## SYNOPSIS

```
use Math::BigInt;
$i = Math::BigInt->new($string);

$i->bneg return BINT               negation
$i->babs return BINT               absolute value
$i->bcmp(BINT) return CODE         compare numbers (undef,<0,=0,>0)
$i->badd(BINT) return BINT         addition
$i->bsub(BINT) return BINT         subtraction
$i->bmul(BINT) return BINT         multiplication
$i->bdiv(BINT) return (BINT,BINT)  division (quo,rem) just quo if scalar
$i->bmod(BINT) return BINT         modulus
$i->bgcd(BINT) return BINT         greatest common divisor
$i->bnorm return BINT              normalization
```

## DESCRIPTION

All basic math operations are overloaded if you declare your big integers as

```
$i = new Math::BigInt '123 456 789 123 456 789';
```

Canonical notation
: Big integer value are strings of the form `/^[+-]\d+$/` with leading zeros suppressed.

Input
: Input values to these routines may be strings of the form `/^\s*[+-]?[\d\s]+$/`.

Output
: Output values always always in canonical form

Actual math is done in an internal format consisting of an array whose first element is the sign (/^[+−]$/) and whose remaining elements are base 100000 digits with the least significant digit first. The string ‘NaN’ is used to represent the result when input arguments are not numbers, as well as the result of dividing by zero.

## EXAMPLES

```
'+0'                             canonical zero value
'   -123 123 123'                canonical value '-123123123'
'1 23 456 7890'                  canonical value '+1234567890'
```

### Autocreating constants

After `use Math::BigInt ':constant'` all the integer decimal constants in the given scope are converted to `Math::BigInt`. This conversion happens at compile time.

In particular

```
perl -MMath::BigInt=:constant -e 'print 2**100'
```

print the integer value of `2**100`. Note that without convertion of constants the expression 2**100 will be calculatted as floating point number.

## BUGS

The current version of this module is a preliminary version of the real thing that is currently (as of perl5.002) under development.

**AUTHOR**

Mark Biggar, overloaded interface by Ilya Zakharevich.

## NAME

Math::Complex − complex numbers and associated mathematical functions

## SYNOPSIS

```
use Math::Complex;

$z = Math::Complex->make(5, 6);
$t = 4 − 3*i + $z;
$j = cplxe(1, 2*pi/3);
```

## DESCRIPTION

This package lets you create and manipulate complex numbers. By default, *Perl* limits itself to real numbers, but an extra `use` statement brings full complex support, along with a full set of mathematical functions typically associated with and/or extended to complex numbers.

If you wonder what complex numbers are, they were invented to be able to solve the following equation:

```
x*x = −1
```

and by definition, the solution is noted *i* (engineers use *j* instead since *i* usually denotes an intensity, but the name does not matter). The number *i* is a pure *imaginary* number.

The arithmetics with pure imaginary numbers works just like you would expect it with real numbers... you just have to remember that

```
i*i = −1
```

so you have:

```
5i + 7i = i * (5 + 7) = 12i
4i − 3i = i * (4 − 3) = i
4i * 2i = −8
6i / 2i = 3
1 / i = −i
```

Complex numbers are numbers that have both a real part and an imaginary part, and are usually noted:

```
a + bi
```

where `a` is the *real* part and `b` is the *imaginary* part. The arithmetic with complex numbers is straightforward. You have to keep track of the real and the imaginary parts, but otherwise the rules used for real numbers just apply:

```
(4 + 3i) + (5 − 2i) = (4 + 5) + i(3 − 2) = 9 + i
(2 + i) * (4 − i) = 2*4 + 4i −2i −i*i = 8 + 2i + 1 = 9 + 2i
```

A graphical representation of complex numbers is possible in a plane (also called the *complex plane*, but it's really a 2D plane). The number

```
z = a + bi
```

is the point whose coordinates are (a, b). Actually, it would be the vector originating from (0, 0) to (a, b). It follows that the addition of two complex numbers is a vectorial addition.

Since there is a bijection between a point in the 2D plane and a complex number (i.e. the mapping is unique and reciprocal), a complex number can also be uniquely identified with polar coordinates:

```
[rho, theta]
```

where `rho` is the distance to the origin, and `theta` the angle between the vector and the *x* axis. There is a notation for this using the exponential form, which is:

```
rho * exp(i * theta)
```

where *i* is the famous imaginary number introduced above. Conversion between this form and the cartesian form `a + bi` is immediate:

```
a = rho * cos(theta)
b = rho * sin(theta)
```

which is also expressed by this formula:

```
z = rho * exp(i * theta) = rho * (cos theta + i * sin theta)
```

In other words, it's the projection of the vector onto the *x* and *y* axes. Mathematicians call *rho* the *norm* or *modulus* and *theta* the *argument* of the complex number. The *norm* of `z` will be noted `abs(z)`.

The polar notation (also known as the trigonometric representation) is much more handy for performing multiplications and divisions of complex numbers, whilst the cartesian notation is better suited for additions and subtractions. Real numbers are on the *x* axis, and therefore *theta* is zero or *pi*.

All the common operations that can be performed on a real number have been defined to work on complex numbers as well, and are merely *extensions* of the operations defined on real numbers. This means they keep their natural meaning when there is no imaginary part, provided the number is within their definition set.

For instance, the `sqrt` routine which computes the square root of its argument is only defined for non-negative real numbers and yields a non-negative real number (it is an application from **R+** to **R+**). If we allow it to return a complex number, then it can be extended to negative real numbers to become an application from **R** to **C** (the set of complex numbers):

```
sqrt(x) = x >= 0 ? sqrt(x) : sqrt(-x)*i
```

It can also be extended to be an application from **C** to **C**, whilst its restriction to **R** behaves as defined above by using the following definition:

```
sqrt(z = [r,t]) = sqrt(r) * exp(i * t/2)
```

Indeed, a negative real number can be noted `[x,pi]` (the modulus *x* is always non-negative, so `[x,pi]` is really `-x`, a negative number) and the above definition states that

```
sqrt([x,pi]) = sqrt(x) * exp(i*pi/2) = [sqrt(x),pi/2] = sqrt(x)*i
```

which is exactly what we had defined for negative real numbers above. The `sqrt` returns only one of the solutions: if you want the both, use the `root` function.

All the common mathematical functions defined on real numbers that are extended to complex numbers share that same property of working *as usual* when the imaginary part is zero (otherwise, it would not be called an extension, would it?).

A *new* operation possible on a complex number that is the identity for real numbers is called the *conjugate*, and is noted with an horizontal bar above the number, or `~z` here.

```
 z = a + bi
~z = a - bi
```

Simple... Now look:

```
z * ~z = (a + bi) * (a - bi) = a*a + b*b
```

We saw that the norm of `z` was noted `abs(z)` and was defined as the distance to the origin, also known as:

```
rho = abs(z) = sqrt(a*a + b*b)
```

so

```
z * ~z = abs(z) ** 2
```

If z is a pure real number (i.e. `b == 0`), then the above yields:

```
a * a = abs(a) ** 2
```

which is true (`abs` has the regular meaning for real number, i.e. stands for the absolute value). This example explains why the norm of `z` is noted `abs(z)`: it extends the `abs` function to complex numbers, yet is the regular `abs` we know when the complex number actually has no imaginary part... This justifies *a posteriori* our use of the `abs` notation for the norm.

**OPERATIONS**

Given the following notations:

```
z1 = a + bi = r1 * exp(i * t1)
z2 = c + di = r2 * exp(i * t2)
z = <any complex or real number>
```

the following (overloaded) operations are supported on complex numbers:

```
z1 + z2 = (a + c) + i(b + d)
z1 - z2 = (a - c) + i(b - d)
z1 * z2 = (r1 * r2) * exp(i * (t1 + t2))
z1 / z2 = (r1 / r2) * exp(i * (t1 - t2))
z1 ** z2 = exp(z2 * log z1)
~z = a - bi
abs(z) = r1 = sqrt(a*a + b*b)
sqrt(z) = sqrt(r1) * exp(i * t/2)
exp(z) = exp(a) * exp(i * b)
log(z) = log(r1) + i*t
sin(z) = 1/2i (exp(i * z1) - exp(-i * z))
cos(z) = 1/2 (exp(i * z1) + exp(-i * z))
atan2(z1, z2) = atan(z1/z2)
```

The following extra operations are supported on both real and complex numbers:

```
Re(z) = a
Im(z) = b
arg(z) = t
abs(z) = r

cbrt(z) = z ** (1/3)
log10(z) = log(z) / log(10)
logn(z, n) = log(z) / log(n)

tan(z) = sin(z) / cos(z)

csc(z) = 1 / sin(z)
sec(z) = 1 / cos(z)
cot(z) = 1 / tan(z)

asin(z) = -i * log(i*z + sqrt(1-z*z))
acos(z) = -i * log(z + i*sqrt(1-z*z))
atan(z) = i/2 * log((i+z) / (i-z))
```

```
acsc(z) = asin(1 / z)
asec(z) = acos(1 / z)
acot(z) = atan(1 / z) = -i/2 * log((i+z) / (z-i))

sinh(z) = 1/2 (exp(z) - exp(-z))
cosh(z) = 1/2 (exp(z) + exp(-z))
tanh(z) = sinh(z) / cosh(z) = (exp(z) - exp(-z)) / (exp(z) + exp(-z))

csch(z) = 1 / sinh(z)
sech(z) = 1 / cosh(z)
coth(z) = 1 / tanh(z)

asinh(z) = log(z + sqrt(z*z+1))
acosh(z) = log(z + sqrt(z*z-1))
atanh(z) = 1/2 * log((1+z) / (1-z))

acsch(z) = asinh(1 / z)
asech(z) = acosh(1 / z)
acoth(z) = atanh(1 / z) = 1/2 * log((1+z) / (z-1))
```

*arg*, *abs*, *log*, *csc*, *cot*, *acsc*, *acot*, *csch*, *coth*, *acosech*, *acotanh*, have aliases *rho*, *theta*, *ln*, *cosec*, *cotan*, *acosec*, *acotan*, *cosech*, *cotanh*, *acosech*, *acotanh*, respectively. `Re`, `Im`, `arg`, `abs`, `rho`, and `theta` can be used also also mutators. The `cbrt` returns only one of the solutions: if you want all three, use the `root` function.

The *root* function is available to compute all the *n* roots of some complex, where *n* is a strictly positive integer. There are exactly *n* such roots, returned as a list. Getting the number mathematicians call `j` such that:

```
1 + j + j*j = 0;
```

is a simple matter of writing:

```
$j = ((root(1, 3))[1];
```

The *k*th root for `z = [r,t]` is given by:

```
(root(z, n))[k] = r**(1/n) * exp(i * (t + 2*k*pi)/n)
```

The *spaceship* comparison operator, <=>, is also defined. In order to ensure its restriction to real numbers is conform to what you would expect, the comparison is run on the real part of the complex number first, and imaginary parts are compared only when the real parts match.

**CREATION**

To create a complex number, use either:

```
$z = Math::Complex->make(3, 4);
$z = cplx(3, 4);
```

if you know the cartesian form of the number, or

```
$z = 3 + 4*i;
```

if you like. To create a number using the polar form, use either:

```
$z = Math::Complex->emake(5, pi/3);
$x = cplxe(5, pi/3);
```

instead. The first argument is the modulus, the second is the angle (in radians, the full circle is 2*pi). (Mnemonic: `e` is used as a notation for complex numbers in the polar form).

It is possible to write:

```
$x = cplxe(-3, pi/4);
```

but that will be silently converted into `[3,-3pi/4]`, since the modulus must be non-negative (it represents the distance to the origin in the complex plane).

It is also possible to have a complex number as either argument of either the `make` or `emake`: the appropriate component of the argument will be used.

```
$z1 = cplx(-2,  1);
$z2 = cplx($z1, 4);
```

**STRINGIFICATION**

When printed, a complex number is usually shown under its cartesian form *a+bi*, but there are legitimate cases where the polar format *[r,t]* is more appropriate.

By calling the routine `Math::Complex::display_format` and supplying either `"polar"` or `"cartesian"`, you override the default display format, which is `"cartesian"`. Not supplying any argument returns the current setting.

This default can be overridden on a per-number basis by calling the `display_format` method instead. As before, not supplying any argument returns the current display format for this number. Otherwise whatever you specify will be the new display format for *this* particular number.

For instance:

```
use Math::Complex;

Math::Complex::display_format('polar');
$j = ((root(1, 3))[1];
print "j = $j\n";               # Prints "j = [1,2pi/3]
$j->display_format('cartesian');
print "j = $j\n";               # Prints "j = -0.5+0.866025403784439i"
```

The polar format attempts to emphasize arguments like *k*pi/n* (where *n* is a positive integer and *k* an integer within [−9,+9]).

**USAGE**

Thanks to overloading, the handling of arithmetics with complex numbers is simple and almost transparent.

Here are some examples:

```
use Math::Complex;

$j = cplxe(1, 2*pi/3);  # $j ** 3 == 1
print "j = $j, j**3 = ", $j ** 3, "\n";
print "1 + j + j**2 = ", 1 + $j + $j**2, "\n";

$z = -16 + 0*i;                 # Force it to be a complex
print "sqrt($z) = ", sqrt($z), "\n";

$k = exp(i * 2*pi/3);
print "$j - $k = ", $j - $k, "\n";

$z->Re(3);                      # Re, Im, arg, abs,
$j->arg(2);                     # (the last two aka rho, theta)
                                # can be used also as mutators.
```

**ERRORS DUE TO DIVISION BY ZERO OR LOGARITHM OF ZERO**

The division (/) and the following functions

```
log     ln      log10   logn
tan     sec     csc     cot
atan    asec    acsc    acot
tanh    sech    csch    coth
atanh   asech   acsch   acoth
```

cannot be computed for all arguments because that would mean dividing by zero or taking logarithm of zero. These situations cause fatal runtime errors looking like this

```
cot(0): Division by zero.
(Because in the definition of cot(0), the divisor sin(0) is 0)
Died at ...
```

or

```
atanh(-1): Logarithm of zero.
Died at...
```

For the `csc`, `cot`, `asec`, `acsc`, `acot`, `csch`, `coth`, `asech`, `acsch`, the argument cannot be `0` (zero). For the the logarithmic functions and the `atanh`, `acoth`, the argument cannot be `1` (one). For the `atanh`, `acoth`, the argument cannot be `-1` (minus one). For the `atan`, `acot`, the argument cannot be `i` (the imaginary unit). For the `atan`, `acoth`, the argument cannot be `-i` (the negative imaginary unit). For the `tan`, `sec`, `tanh`, the argument cannot be *pi/2 + k * pi*, where *k* is any integer.

Note that because we are operating on approximations of real numbers, these errors can happen when merely 'too close' to the singularities listed above. For example `tan(2*atan2(1,1)+1e-15)` will die of division by zero.

**ERRORS DUE TO INDIGESTIBLE ARGUMENTS**

The `make` and `emake` accept both real and complex arguments. When they cannot recognize the arguments they will die with error messages like the following

```
Math::Complex::make: Cannot take real part of ...
Math::Complex::make: Cannot take real part of ...
Math::Complex::emake: Cannot take rho of ...
Math::Complex::emake: Cannot take theta of ...
```

**BUGS**

Saying `use Math::Complex;` exports many mathematical routines in the caller environment and even overrides some (`sqrt`, `log`). This is construed as a feature by the Authors, actually... ;–)

All routines expect to be given real or complex numbers. Don't attempt to use BigFloat, since Perl has currently no rule to disambiguate a '+' operation (for instance) between two overloaded entities.

In Cray UNICOS there is some strange numerical instability that results in *root()*, *cos()*, *sin()*, *cosh()*, *sinh()*, losing accuracy fast. Beware. The bug may be in UNICOS math libs, in UNICOS C compiler, in Math::Complex. Whatever it is, it does not manifest itself anywhere else where Perl runs.

**AUTHORS**

Raphael Manfredi <*Raphael_Manfredi@grenoble.hp.com*> and Jarkko Hietaniemi <*jhi@iki.fi*>.

Extensive patches by Daniel S. Lewart <*d-lewart@uiuc.edu*>.

## NAME

Math::Trig – trigonometric functions

## SYNOPSIS

```
use Math::Trig;

$x = tan(0.9);
$y = acos(3.7);
$z = asin(2.4);

$halfpi = pi/2;

$rad = deg2rad(120);
```

## DESCRIPTION

`Math::Trig` defines many trigonometric functions not defined by the core Perl which defines only the `sin()` and `cos()`. The constant **pi** is also defined as are a few convenience functions for angle conversions.

## TRIGONOMETRIC FUNCTIONS

The tangent

**tan**

The cofunctions of the sine, cosine, and tangent (cosec/csc and cotan/cot are aliases)

**csc**, **cosec**, **sec**, **sec**, **cot**, **cotan**

The arcus (also known as the inverse) functions of the sine, cosine, and tangent

**asin**, **acos**, **atan**

The principal value of the arc tangent of y/x

**atan2**(y, x)

The arcus cofunctions of the sine, cosine, and tangent (acosec/acsc and acotan/acot are aliases)

**acsc**, **acosec**, **asec**, **acot**, **acotan**

The hyperbolic sine, cosine, and tangent

**sinh**, **cosh**, **tanh**

The cofunctions of the hyperbolic sine, cosine, and tangent (cosech/csch and cotanh/coth are aliases)

**csch**, **cosech**, **sech**, **coth**, **cotanh**

The arcus (also known as the inverse) functions of the hyperbolic sine, cosine, and tangent

**asinh**, **acosh**, **atanh**

The arcus cofunctions of the hyperbolic sine, cosine, and tangent (acsch/acosech and acoth/acotanh are aliases)

**acsch**, **acosech**, **asech**, **acoth**, **acotanh**

The trigonometric constant **pi** is also defined.

$pi2 = 2 * **pi**;

### ERRORS DUE TO DIVISION BY ZERO

The following functions

```
acoth
acsc
acsch
asec
asech
atanh
cot
coth
csc
csch
sec
sech
tan
tanh
```

cannot be computed for all arguments because that would mean dividing by zero or taking logarithm of zero. These situations cause fatal runtime errors looking like this

```
cot(0): Division by zero.
(Because in the definition of cot(0), the divisor sin(0) is 0)
Died at ...
```

or

```
atanh(-1): Logarithm of zero.
Died at...
```

For the `csc`, `cot`, `asec`, `acsc`, `acot`, `csch`, `coth`, `asech`, `acsch`, the argument cannot be `0` (zero). For the `atanh`, `acoth`, the argument cannot be `1` (one). For the `atanh`, `acoth`, the argument cannot be `-1` (minus one). For the `tan`, `sec`, `tanh`, `sech`, the argument cannot be *pi/2 + k * pi*, where *k* is any integer.

### SIMPLE (REAL) ARGUMENTS, COMPLEX RESULTS

Please note that some of the trigonometric functions can break out from the **real axis** into the **complex plane**. For example `asin(2)` has no definition for plain real numbers but it has definition for complex numbers.

In Perl terms this means that supplying the usual Perl numbers (also known as scalars, please see the *perldata* manpage) as input for the trigonometric functions might produce as output results that no more are simple real numbers: instead they are complex numbers.

The `Math::Trig` handles this by using the `Math::Complex` package which knows how to handle complex numbers, please see the *Math::Complex* manpage for more information. In practice you need not to worry about getting complex numbers as results because the `Math::Complex` takes care of details like for example how to display complex numbers. For example:

```
print asin(2), "\n";
```

`should produce something like this (take or leave few last decimals):`

```
1.5707963267949-1.31695789692482i
```

That is, a complex number with the real part of approximately `1.571` and the imaginary part of approximately `-1.317`.

## PLANE ANGLE CONVERSIONS

(Plane, 2-dimensional) angles may be converted with the following functions.

```
$radians  = deg2rad($degrees);
$radians  = grad2rad($gradians);

$degrees  = rad2deg($radians);
$degrees  = grad2deg($gradians);

$gradians = deg2grad($degrees);
$gradians = rad2grad($radians);
```

The full circle is 2 *pi* radians or *360* degrees or *400* gradians.

## RADIAL COORDINATE CONVERSIONS

**Radial coordinate systems** are the **spherical** and the **cylindrical** systems, explained shortly in more detail.

You can import radial coordinate conversion functions by using the `:radial` tag:

```
use Math::Trig ':radial';

($rho, $theta, $z)     = cartesian_to_cylindrical($x, $y, $z);
($rho, $theta, $phi)   = cartesian_to_spherical($x, $y, $z);
($x, $y, $z)           = cylindrical_to_cartesian($rho, $theta, $z);
($rho_s, $theta, $phi) = cylindrical_to_spherical($rho_c, $theta, $z);
($x, $y, $z)           = spherical_to_cartesian($rho, $theta, $phi);
($rho_c, $theta, $z)   = spherical_to_cylindrical($rho_s, $theta, $phi);
```

**All angles are in radians**.

### COORDINATE SYSTEMS

**Cartesian** coordinates are the usual rectangular *(x, y, z)*–coordinates.

Spherical coordinates, *(rho, theta, pi)*, are three-dimensional coordinates which define a point in three-dimensional space. They are based on a sphere surface. The radius of the sphere is **rho**, also known as the *radial* coordinate. The angle in the *xy*–plane (around the *z*–axis) is **theta**, also known as the *azimuthal* coordinate. The angle from the *z*–axis is **phi**, also known as the *polar* coordinate. The 'North Pole' is therefore *0, 0, rho*, and the 'Bay of Guinea' (think of the missing big chunk of Africa) *0, pi/2, rho*.

**Beware**: some texts define *theta* and *phi* the other way round, some texts define the *phi* to start from the horizontal plane, some texts use *r* in place of *rho*.

Cylindrical coordinates, *(rho, theta, z)*, are three-dimensional coordinates which define a point in three-dimensional space. They are based on a cylinder surface. The radius of the cylinder is **rho**, also known as the *radial* coordinate. The angle in the *xy*–plane (around the *z*–axis) is **theta**, also known as the *azimuthal* coordinate. The third coordinate is the *z*, pointing up from the **theta**–plane.

### 3-D ANGLE CONVERSIONS

Conversions to and from spherical and cylindrical coordinates are available. Please notice that the conversions are not necessarily reversible because of the equalities like *pi* angles being equal to −*pi* angles.

cartesian_to_cylindrical

```
($rho, $theta, $z) = cartesian_to_cylindrical($x, $y, $z);
```

cartesian_to_spherical

```
($rho, $theta, $phi) = cartesian_to_spherical($x, $y, $z);
```

cylindrical_to_cartesian

```
($x, $y, $z) = cylindrical_to_cartesian($rho, $theta, $z);
```

cylindrical_to_spherical

```
($rho_s, $theta, $phi) = cylindrical_to_spherical($rho_c, $theta, $z);
```

Notice that when `$z` is not 0 `$rho_s` is not equal to `$rho_c`.

spherical_to_cartesian

```
($x, $y, $z) = spherical_to_cartesian($rho, $theta, $phi);
```

spherical_to_cylindrical

```
($rho_c, $theta, $z) = spherical_to_cylindrical($rho_s, $theta, $phi);
```

Notice that when `$z` is not 0 `$rho_c` is not equal to `$rho_s`.

## GREAT CIRCLE DISTANCES

You can compute spherical distances, called **great circle distances**, by importing the `great_circle_distance` function:

```
use Math::Trig 'great_circle_distance'

$distance = great_circle_distance($theta0, $phi0, $theta1, $phi, [, $rho]);
```

The *great circle distance* is the shortest distance between two points on a sphere. The distance is in `$rho` units. The `$rho` is optional, it defaults to 1 (the unit sphere), therefore the distance defaults to radians.

## EXAMPLES

To calculate the distance between London (51.3N 0.5W) and Tokyo (35.7N 139.8E) in kilometers:

```
use Math::Trig qw(great_circle_distance deg2rad);

# Notice the 90 - latitude: phi zero is at the North Pole.
@L = (deg2rad(-0.5), deg2rad(90 - 51.3));
@T = (deg2rad(139.8),deg2rad(90 - 35.7));

$km = great_circle_distance(@L, @T, 6378);
```

The answer may be off by up to 0.3% because of the irregular (slightly aspherical) form of the Earth.

## BUGS

Saying `use Math::Trig;` exports many mathematical routines in the caller environment and even overrides some (`sin`, `cos`). This is construed as a feature by the Authors, actually... ;–)

The code is not optimized for speed, especially because we use `Math::Complex` and thus go quite near complex numbers while doing the computations even when the arguments are not. This, however, cannot be completely avoided if we want things like `asin(2)` to give an answer instead of giving a fatal runtime error.

## AUTHORS

Jarkko Hietaniemi <*jhi@iki.fi*> and Raphael Manfredi <*Raphael_Manfredi@grenoble.hp.com*>.

**NAME**

NDBM_File – Tied access to ndbm files

**SYNOPSIS**

```
use NDBM_File;
use Fcntl;       # for O_ constants

tie(%h, 'NDBM_File', 'Op.dbmx', O_RDWR|O_CREAT, 0640);

untie %h;
```

**DESCRIPTION**

See the `tie` entry in the *perlfunc* manpage

## NAME

Net::hostent – by-name interface to Perl's built-in gethost*() functions

## SYNOPSIS

```
use Net::hostnet;
```

## DESCRIPTION

This module's default exports override the core *gethostbyname()* and *gethostbyaddr()* functions, replacing them with versions that return "Net::hostent" objects. This object has methods that return the similarly named structure field name from the C's hostent structure from *netdb.h*; namely name, aliases, addrtype, length, and addr_list. The aliases and addr_list methods return array reference, the rest scalars. The addr method is equivalent to the zeroth element in the addr_list array reference.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `h_`. Thus, `$host_obj->name()` corresponds to `$h_name` if you import the fields. Array references are available as regular array variables, so for example `@{ $host_obj->aliases() }` would be simply `@h_aliases`.

The *gethost()* funtion is a simple front-end that forwards a numeric argument to *gethostbyaddr()* by way of Socket::inet_aton, and the rest to *gethostbyname()*.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

## EXAMPLES

```
use Net::hostent;
use Socket;

@ARGV = ('netscape.com') unless @ARGV;

for $host ( @ARGV ) {

   unless ($h = gethost($host)) {
       warn "$0: no such host: $host\n";
       next;
   }

   printf "\n%s is %s%s\n",
           $host,
           lc($h->name) eq lc($host) ? "" : "*really* ",
           $h->name;

   print "\taliases are ", join(", ", @{$h->aliases}), "\n"
               if @{$h->aliases};

   if ( @{$h->addr_list} > 1 ) {
       my $i;
       for $addr ( @{$h->addr_list} ) {
           printf "\taddr #%d is [%s]\n", $i++, inet_ntoa($addr);
       }
   } else {
       printf "\taddress is [%s]\n", inet_ntoa($h->addr);
   }
```

```
        if ($h = gethostbyaddr($h->addr)) {
            if (lc($h->name) ne lc($host)) {
                printf "\tThat addr reverses to host %s!\n", $h->name;
                $host = $h->name;
                redo;
            }
        }
    }
```

**NOTE**

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

**AUTHOR**

Tom Christiansen

## NAME

Net::netent – by-name interface to Perl's built-in getnet*() functions

## SYNOPSIS

```
use Net::netent qw(:FIELDS);
getnetbyname("loopback")               or die "bad net";
printf "%s is %08X\n", $n_name, $n_net;

use Net::netent;

$n = getnetbyname("loopback")          or die "bad net";
{ # there's gotta be a better way, eh?
    @bytes = unpack("C4", pack("N", $n->net));
    shift @bytes while @bytes && $bytes[0] == 0;
}
printf "%s is %08X [%d.%d.%d.%d]\n", $n->name, $n->net, @bytes;
```

## DESCRIPTION

This module's default exports override the core *getnetbyname()* and *getnetbyaddr()* functions, replacing them with versions that return "Net::netent" objects. This object has methods that return the similarly named structure field name from the C's netent structure from *netdb.h*; namely name, aliases, addrtype, and net. The aliases method returns an array reference, the rest scalars.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `n_`. Thus, `$net_obj->name()` corresponds to `$n_name` if you import the fields. Array references are available as regular array variables, so for example `@{ $net_obj->aliases() }` would be simply `@n_aliases`.

The *getnet()* funtion is a simple front-end that forwards a numeric argument to *getnetbyaddr()*, and the rest to *getnetbyname()*.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

## EXAMPLES

The *getnet()* functions do this in the Perl core:

```
sv_setiv(sv, (I32)nent->n_net);
```

The *gethost()* functions do this in the Perl core:

```
sv_setpvn(sv, hent->h_addr, len);
```

That means that the address comes back in binary for the host functions, and as a regular perl integer for the net ones. This seems a bug, but here's how to deal with it:

```
use strict;
use Socket;
use Net::netent;

@ARGV = ('loopback') unless @ARGV;

my($n, $net);

for $net ( @ARGV ) {

    unless ($n = getnetbyname($net)) {
       warn "$0: no such net: $net\n";
       next;
    }

    printf "\n%s is %s%s\n",
           $net,
           lc($n->name) eq lc($net) ? "" : "*really* ",
           $n->name;

    print "\taliases are ", join(", ", @{$n->aliases}), "\n"
               if @{$n->aliases};

    # this is stupid; first, why is this not in binary?
    # second, why am i going through these convolutions
    # to make it looks right
    {
       my @a = unpack("C4", pack("N", $n->net));
       shift @a while @a && $a[0] == 0;
       printf "\taddr is %s [%d.%d.%d.%d]\n", $n->net, @a;
    }

    if ($n = getnetbyaddr($n->net)) {
       if (lc($n->name) ne lc($net)) {
           printf "\tThat addr reverses to net %s!\n", $n->name;
           $net = $n->name;
           redo;
       }
    }
}
```

**NOTE**

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

**AUTHOR**

Tom Christiansen

**NAME**

Net::Ping − check a remote host for reachability

**SYNOPSIS**

```
use Net::Ping;

$p = Net::Ping->new();
print "$host is alive.\n" if $p->ping($host);
$p->close();

$p = Net::Ping->new("icmp");
foreach $host (@host_array)
{
    print "$host is ";
    print "NOT " unless $p->ping($host, 2);
    print "reachable.\n";
    sleep(1);
}
$p->close();

$p = Net::Ping->new("tcp", 2);
while ($stop_time > time())
{
    print "$host not reachable ", scalar(localtime()), "\n"
        unless $p->ping($host);
    sleep(300);
}
undef($p);

# For backward compatibility
print "$host is alive.\n" if pingecho($host);
```

**DESCRIPTION**

This module contains methods to test the reachability of remote hosts on a network. A ping object is first created with optional parameters, a variable number of hosts may be pinged multiple times and then the connection is closed.

You may choose one of three different protocols to use for the ping. With the "tcp" protocol the *ping()* method attempts to establish a connection to the remote host's echo port. If the connection is successfully established, the remote host is considered reachable. No data is actually echoed. This protocol does not require any special privileges but has higher overhead than the other two protocols.

Specifying the "udp" protocol causes the *ping()* method to send a udp packet to the remote host's echo port. If the echoed packet is received from the remote host and the received packet contains the same data as the packet that was sent, the remote host is considered reachable. This protocol does not require any special privileges.

If the "icmp" protocol is specified, the *ping()* method sends an icmp echo message to the remote host, which is what the UNIX ping program does. If the echoed message is received from the remote host and the echoed information is correct, the remote host is considered reachable. Specifying the "icmp" protocol requires that the program be run as root or that the program be setuid to root.

### Functions

Net::Ping->new([$proto [, $def_timeout [, $bytes]]]);
: Create a new ping object. All of the parameters are optional. `$proto` specifies the protocol to use when doing a ping. The current choices are ''tcp'', ''udp'' or ''icmp''. The default is ''udp''.

  If a default timeout ($def_timeout) in seconds is provided, it is used when a timeout is not given to the *ping()* method (below). The timeout must be greater than 0 and the default, if not specified, is 5 seconds.

  If the number of data bytes ($bytes) is given, that many data bytes are included in the ping packet sent to the remote host. The number of data bytes is ignored if the protocol is ''tcp''. The minimum (and default) number of data bytes is 1 if the protocol is ''udp'' and 0 otherwise. The maximum number of data bytes that can be specified is 1024.

$p->ping($host [, $timeout]);
: Ping the remote host and wait for a response. `$host` can be either the hostname or the IP number of the remote host. The optional timeout must be greater than 0 seconds and defaults to whatever was specified when the ping object was created. If the hostname cannot be found or there is a problem with the IP number, undef is returned. Otherwise, 1 is returned if the host is reachable and 0 if it is not. For all practical purposes, undef and 0 and can be treated as the same case.

$p->close();
: Close the network connection for this ping object. The network connection is also closed by ''undef `$p`''. The network connection is automatically closed if the ping object goes out of scope (e.g. `$p` is local to a subroutine and you leave the subroutine).

pingecho($host [, $timeout]);
: To provide backward compatibility with the previous version of Net::Ping, a *pingecho()* subroutine is available with the same functionality as before. *pingecho()* uses the tcp protocol. The return values and parameters are the same as described for the *ping()* method. This subroutine is obsolete and may be removed in a future version of Net::Ping.

## WARNING

*pingecho()* or a ping object with the tcp protocol use *alarm()* to implement the timeout. So, don't use *alarm()* in your program while you are using *pingecho()* or a ping object with the tcp protocol. The udp and icmp protocols do not use *alarm()* to implement the timeout.

## NOTES

There will be less network overhead (and some efficiency in your program) if you specify either the udp or the icmp protocol. The tcp protocol will generate 2.5 times or more traffic for each ping than either udp or icmp. If many hosts are pinged frequently, you may wish to implement a small wait (e.g. 25ms or more) between each ping to avoid flooding your network with packets.

The icmp protocol requires that the program be run as root or that it be setuid to root. The tcp and udp protocols do not require special privileges, but not all network devices implement the echo protocol for tcp or udp.

Local hosts should normally respond to pings within milliseconds. However, on a very congested network it may take up to 3 seconds or longer to receive an echo packet from the remote host. If the timeout is set too low under these conditions, it will appear that the remote host is not reachable (which is almost the truth).

Reachability doesn't necessarily mean that the remote host is actually functioning beyond its ability to echo packets.

Because of a lack of anything better, this module uses its own routines to pack and unpack ICMP packets. It would be better for a separate module to be written which understands all of the different kinds of ICMP packets.

## NAME

Net::protoent − by-name interface to Perl's built-in getproto*() functions

## SYNOPSIS

```
use Net::protoent;
$p = getprotobyname(shift || 'tcp') || die "no proto";
printf "proto for %s is %d, aliases are %s\n",
   $p->name, $p->proto, "@{$p->aliases}";

use Net::protoent qw(:FIELDS);
getprotobyname(shift || 'tcp') || die "no proto";
print "proto for $p_name is $p_proto, aliases are @p_aliases\n";
```

## DESCRIPTION

This module's default exports override the core *getprotoent()*, *getprotobyname()*, and *getnetbyport()* functions, replacing them with versions that return "Net::protoent" objects. They take default second arguments of "tcp". This object has methods that return the similarly named structure field name from the C's protoent structure from *netdb.h*; namely name, aliases, and proto. The aliases method returns an array reference, the rest scalars.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `p_`. Thus, `$proto_obj->name()` corresponds to `$p_name` if you import the fields. Array references are available as regular array variables, so for example `@{ $proto_obj->aliases() }` would be simply `@p_aliases`.

The *getproto()* function is a simple front-end that forwards a numeric argument to *getprotobyport()*, and the rest to *getprotobyname()*.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

## NOTE

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

## AUTHOR

Tom Christiansen

**NAME**

Net::servent − by-name interface to Perl's built-in getserv*() functions

**SYNOPSIS**

```
use Net::servent;
$s = getservbyname(shift || 'ftp') || die "no service";
printf "port for %s is %s, aliases are %s\n",
   $s->name, $s->port, "@{$s->aliases}";

use Net::servent qw(:FIELDS);
getservbyname(shift || 'ftp') || die "no service";
print "port for $s_name is $s_port, aliases are @s_aliases\n";
```

**DESCRIPTION**

This module's default exports override the core *getservent()*, *getservbyname()*, and *getnetbyport()* functions, replacing them with versions that return "Net::servent" objects. They take default second arguments of "tcp". This object has methods that return the similarly named structure field name from the C's servent structure from *netdb.h*; namely name, aliases, port, and proto. The aliases method returns an array reference, the rest scalars.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `n_`. Thus, `$serv_obj->name()` corresponds to `$s_name` if you import the fields. Array references are available as regular array variables, so for example `@{ $serv_obj->aliases() }` would be simply `@s_aliases`.

The *getserv()* function is a simple front-end that forwards a numeric argument to *getservbyport()*, and the rest to *getservbyname()*.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

**EXAMPLES**

```
use Net::servent qw(:FIELDS);

while (@ARGV) {
    my ($service, $proto) = ((split m!/!, shift), 'tcp');
    my $valet = getserv($service, $proto);
    unless ($valet) {
        warn "$0: No service: $service/$proto\n"
        next;
    }
    printf "service $service/$proto is port %d\n", $valet->port;
    print "alias are @s_aliases\n" if @s_aliases;
}
```

**NOTE**

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

**AUTHOR**

Tom Christiansen

## NAME

O − Generic interface to Perl Compiler backends

## SYNOPSIS

```
        perl -MO=Backend[,OPTIONS] foo.pl
```

## DESCRIPTION

This is the module that is used as a frontend to the Perl Compiler.

## CONVENTIONS

Most compiler backends use the following conventions: OPTIONS consists of a comma-separated list of words (no white-space). The `−v` option usually puts the backend into verbose mode. The `−ofile` option generates output to **file** instead of stdout. The `−D` option followed by various letters turns on various internal debugging flags. See the documentation for the desired backend (named `B::Backend` for the example above) to find out about that backend.

## IMPLEMENTATION

This section is only necessary for those who want to write a compiler backend module that can be used via this module.

The command-line mentioned in the SYNOPSIS section corresponds to the Perl code

```
    use O ("Backend", OPTIONS);
```

The `import` function which that calls loads in the appropriate `B::Backend` module and calls the `compile` function in that package, passing it OPTIONS. That function is expected to return a sub reference which we'll call CALLBACK. Next, the ''compile-only'' flag is switched on (equivalent to the command-line option `−c`) and an END block is registered which calls CALLBACK. Thus the main Perl program mentioned on the command-line is read in, parsed and compiled into internal syntax tree form. Since the `−c` flag is set, the program does not start running (excepting BEGIN blocks of course) but the CALLBACK function registered by the compiler backend is called.

In summary, a compiler backend module should be called ''B::Foo'' for some foo and live in the appropriate directory for that name. It should define a function called `compile`. When the user types

```
    perl -MO=Foo,OPTIONS foo.pl
```

that function is called and is passed those OPTIONS (split on commas). It should return a sub ref to the main compilation function. After the user's program is loaded and parsed, that returned sub ref is invoked which can then go ahead and do the compilation, usually by making use of the `B` module's functionality.

## AUTHOR

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

**NAME**

Opcode − Disable named opcodes when compiling perl code

**SYNOPSIS**

```
use Opcode;
```

**DESCRIPTION**

Perl code is always compiled into an internal format before execution.

Evaluating perl code (e.g. via ‘‘eval’’ or ‘‘do ‘file’"') causes the code to be compiled into an internal format and then, provided there was no error in the compilation, executed. The internal format is based on many distinct *opcodes*.

By default no opmask is in effect and any code can be compiled.

The Opcode module allow you to define an *operator mask* to be in effect when perl *next* compiles any code. Attempting to compile code which contains a masked opcode will cause the compilation to fail with an error. The code will not be executed.

**NOTE**

The Opcode module is not usually used directly. See the ops pragma and Safe modules for more typical uses.

**WARNING**

The authors make **no warranty**, implied or otherwise, about the suitability of this software for safety or security purposes.

The authors shall not in any case be liable for special, incidental, consequential, indirect or other similar damages arising from the use of this software.

Your mileage will vary. If in any doubt **do not use it**.

**Operator Names and Operator Lists**

The canonical list of operator names is the contents of the array op_name defined and initialised in file *opcode.h* of the Perl source distribution (and installed into the perl library).

Each operator has both a terse name (its opname) and a more verbose or recognisable descriptive name. The opdesc function can be used to return a list of descriptions for a list of operators.

Many of the functions and methods listed below take a list of operators as parameters. Most operator lists can be made up of several types of element. Each element can be one of

an operator name (opname)

Operator names are typically small lowercase words like enterloop, leaveloop, last, next, redo etc. Sometimes they are rather cryptic like gv2cv, i_ncmp and ftsvtx.

an operator tag name (optag)

Operator tags can be used to refer to groups (or sets) of operators. Tag names always begin with a colon. The Opcode module defines several optags and the user can define others using the define_optag function.

a negated opname or optag

An opname or optag can be prefixed with an exclamation mark, e.g., !mkdir. Negating an opname or optag means remove the corresponding ops from the accumulated set of ops at that point.

an operator set (opset)

An *opset* as a binary string of approximately 43 bytes which holds a set or zero or more operators.

The opset and opset_to_ops functions can be used to convert from a list of operators to an opset and *vice versa*.

Wherever a list of operators can be given you can use one or more opsets. See also Manipulating

Opsets below.

**Opcode Functions**

The Opcode package contains functions for manipulating operator names tags and sets. All are available for export by the package.

opcodes
In a scalar context opcodes returns the number of opcodes in this version of perl (around 340 for perl5.002).

In a list context it returns a list of all the operator names. (Not yet implemented, use `@names` = *opset_to_ops* (full_opset).)

opset (OP, ...)
Returns an opset containing the listed operators.

opset_to_ops (OPSET)
Returns a list of operator names corresponding to those operators in the set.

opset_to_hex (OPSET)
Returns a string representation of an opset. Can be handy for debugging.

full_opset
Returns an opset which includes all operators.

empty_opset
Returns an opset which contains no operators.

invert_opset (OPSET)
Returns an opset which is the inverse set of the one supplied.

verify_opset (OPSET, ...)
Returns true if the supplied opset looks like a valid opset (is the right length etc) otherwise it returns false. If an optional second parameter is true then verify_opset will croak on an invalid opset instead of returning false.

Most of the other Opcode functions call verify_opset automatically and will croak if given an invalid opset.

define_optag (OPTAG, OPSET)
Define OPTAG as a symbolic name for OPSET. Optag names always start with a colon **:**.

The optag name used must not be defined already (define_optag will croak if it is already defined). Optag names are global to the perl process and optag definitions cannot be altered or deleted once defined.

It is strongly recommended that applications using Opcode should use a leading capital letter on their tag names since lowercase names are reserved for use by the Opcode module. If using Opcode within a module you should prefix your tags names with the name of your module to ensure uniqueness and thus avoid clashes with other modules.

opmask_add (OPSET)
Adds the supplied opset to the current opmask. Note that there is currently *no* mechanism for unmasking ops once they have been masked. This is intentional.

opmask
Returns an opset corresponding to the current opmask.

opdesc (OP, ...)
This takes a list of operator names and returns the corresponding list of operator descriptions.

opdump (PAT)
Dumps to STDOUT a two column list of op names and op descriptions. If an optional pattern is given then only lines which match the (case insensitive) pattern will be output.

It's designed to be used as a handy command line utility:

```
perl -MOpcode=opdump -e opdump
perl -MOpcode=opdump -e 'opdump Eval'
```

**Manipulating Opsets**

Opsets may be manipulated using the perl bit vector operators & (and), | (or), ^ (xor) and ~ (negate/invert).

However you should never rely on the numerical position of any opcode within the opset. In other words both sides of a bit vector operator should be opsets returned from Opcode functions.

Also, since the number of opcodes in your current version of perl might not be an exact multiple of eight, there may be unused bits in the last byte of an upset. This should not cause any problems (Opcode functions ignore those extra bits) but it does mean that using the ~ operator will typically not produce the same 'physical' opset 'string' as the invert_opset function.

**TO DO (maybe)**

```
$bool = opset_eq($opset1, $opset2)  true if opsets are logically eqiv

$yes = opset_can($opset, @ops)      true if $opset has all @ops set

@diff = opset_diff($opset1, $opset2) => ('foo', '!bar', ...)
```

**Predefined Opcode Tags**

:base_core

```
null stub scalar pushmark wantarray const defined undef

rv2sv sassign

rv2av aassign aelem aelemfast aslice av2arylen

rv2hv helem hslice each values keys exists delete

preinc i_preinc predec i_predec postinc i_postinc postdec i_postdec
int hex oct abs pow multiply i_multiply divide i_divide
modulo i_modulo add i_add subtract i_subtract

left_shift right_shift bit_and bit_xor bit_or negate i_negate
not complement

lt i_lt gt i_gt le i_le ge i_ge eq i_eq ne i_ne ncmp i_ncmp
slt sgt sle sge seq sne scmp

substr vec stringify study pos length index rindex ord chr

ucfirst lcfirst uc lc quotemeta trans chop schop chomp schomp

match split qr

list lslice splice push pop shift unshift reverse

cond_expr flip flop andassign orassign and or xor

warn die lineseq nextstate unstack scope enter leave

rv2cv anoncode prototype
```

```
entersub leavesub return method -- XXX loops via recursion?

leaveeval -- needed for Safe to operate, is safe without entereval
```

:base_mem
These memory related ops are not included in :base_core because they can easily be used to implement a resource attack (e.g., consume all available memory).

```
concat repeat join range

anonlist anonhash
```

Note that despite the existance of this optag a memory resource attack may still be possible using only :base_core ops.

Disabling these ops is a *very* heavy handed way to attempt to prevent a memory resource attack. It's probable that a specific memory limit mechanism will be added to perl in the near future.

:base_loop
These loop ops are not included in :base_core because they can easily be used to implement a resource attack (e.g., consume all available CPU time).

```
grepstart grepwhile
mapstart mapwhile
enteriter iter
enterloop leaveloop
last next redo
goto
```

:base_io
These ops enable *filehandle* (rather than filename) based input and output. These are safe on the assumption that only pre-existing filehandles are available for use. To create new filehandles other ops such as open would need to be enabled.

```
readline rcatline getc read

formline enterwrite leavewrite

print sysread syswrite send recv

eof tell seek sysseek

readdir telldir seekdir rewinddir
```

:base_orig
These are a hotchpotch of opcodes still waiting to be considered

```
gvsv gv gelem

padsv padav padhv padany

rv2gv refgen srefgen ref

bless -- could be used to change ownership of objects (reblessing)
```

```
pushre regcmaybe regcreset regcomp subst substcont

sprintf prtf -- can core dump

crypt

tie untie

dbmopen dbmclose
sselect select
pipe_op sockpair

getppid getpgrp setpgrp getpriority setpriority localtime gmtime

entertry leavetry -- can be used to 'hide' fatal errors
```

:base_math
These ops are not included in :base_core because of the risk of them being used to generate floating point exceptions (which would have to be caught using a `$SIG`{FPE} handler).

```
atan2 sin cos exp log sqrt
```

These ops are not included in :base_core because they have an effect beyond the scope of the compartment.

```
rand srand
```

:base_thread
These ops are related to multi-threading.

```
lock threadsv
```

:default
A handy tag name for a *reasonable* default set of ops. (The current ops allowed are unstable while development continues. It will change.)

```
:base_core :base_mem :base_loop :base_io :base_orig :base_thread
```

If safety matters to you (and why else would you be using the Opcode module?) then you should not rely on the definition of this, or indeed any other, optag!

:filesys_read

```
stat lstat readlink

ftatime ftblk ftchr ftctime ftdir fteexec fteowned fteread
ftewrite ftfile ftis ftlink ftmtime ftpipe ftrexec ftrowned
ftrread ftsgid ftsize ftsock ftsuid fttty ftzero ftrwrite ftsvtx

fttext ftbinary

fileno
```

:sys_db

```
ghbyname ghbyaddr ghostent shostent ehostent      -- hosts
gnbyname gnbyaddr gnetent snetent enetent         -- networks
gpbyname gpbynumber gprotoent sprotoent eprotoent -- protocols
gsbyname gsbyport gservent sservent eservent      -- services

gpwnam gpwuid gpwent spwent epwent getlogin       -- users
ggrnam ggrgid ggrent sgrent egrent                -- groups
```

:browse

A handy tag name for a *reasonable* default set of ops beyond the :default optag. Like :default (and indeed all the other optags) its current definition is unstable while development continues. It will change.

The :browse tag represents the next step beyond :default. It it a superset of the :default ops and adds :filesys_read the :sys_db. The intent being that scripts can access more (possibly sensitive) information about your system but not be able to change it.

```
:default :filesys_read :sys_db
```

:filesys_open

```
sysopen open close
umask binmode

open_dir closedir -- other dir ops are in :base_io
```

:filesys_write

```
link unlink rename symlink truncate

mkdir rmdir

utime chmod chown

fcntl -- not strictly filesys related, but possibly as dangerous?
```

:subprocess

```
backtick system

fork

wait waitpid

glob -- access to Cshell via <`rm *`>
```

:ownprocess

```
exec exit kill

time tms -- could be used for timing attacks (paranoid?)
```

:others

This tag holds groups of assorted specialist opcodes that don't warrant having optags defined for them.

SystemV Interprocess Communications:

```
msgctl msgget msgrcv msgsnd

semctl semget semop

shmctl shmget shmread shmwrite
```

:still_to_be_decided

```
chdir
flock ioctl

socket getpeername ssockopt
bind connect listen accept shutdown gsockopt getsockname

sleep alarm -- changes global timer state and signal handling
sort -- assorted problems including core dumps
tied -- can be used to access object implementing a tie
pack unpack -- can be used to create/use memory pointers

entereval -- can be used to hide code from initial compile
require dofile

caller -- get info about calling environment and args

reset

dbstate -- perl -d version of nextstate(ment) opcode
```

:dangerous

This tag is simply a bucket for opcodes that are unlikely to be used via a tag name but need to be tagged for completness and documentation.

```
syscall dump chroot
```

**SEE ALSO**

*ops* (3) -- perl pragma interface to Opcode module.

*Safe* (3) -- Opcode and namespace limited execution compartments

**AUTHORS**

Originally designed and implemented by Malcolm Beattie, mbeattie@sable.ox.ac.uk as part of Safe version 1.

Split out from Safe module version 1, named opcode tags and other changes added by Tim Bunce.

## NAME

Pod::Html − module to convert pod files to HTML

## SYNOPSIS

```
use Pod::Html;
pod2html([options]);
```

## DESCRIPTION

Converts files from pod format (see the *perlpod* manpage) to HTML format. It can automatically generate indexes and cross-references, and it keeps a cache of things it knows how to cross-reference.

## ARGUMENTS

Pod::Html takes the following arguments:

help

```
--help
```

Displays the usage message.

htmlroot

```
--htmlroot=name
```

Sets the base URL for the HTML files. When cross-references are made, the HTML root is prepended to the URL.

infile

```
--infile=name
```

Specify the pod file to convert. Input is taken from STDIN if no infile is specified.

outfile

```
--outfile=name
```

Specify the HTML file to create. Output goes to STDOUT if no outfile is specified.

podroot

```
--podroot=name
```

Specify the base directory for finding library pods.

podpath

```
--podpath=name:...:name
```

Specify which subdirectories of the podroot contain pod files whose HTML converted forms can be linked-to in cross-references.

libpods

```
--libpods=name:...:name
```

List of page names (eg, "perlfunc") which contain linkable `=item`s.

netscape

```
--netscape
```

Use Netscape HTML directives when applicable.

nonetscape

```
--nonetscape
```

Do not use Netscape HTML directives (default).

index

```
--index
```

Generate an index at the top of the HTML file (default behaviour).

noindex

```
--noindex
```

Do not generate an index at the top of the HTML file.

recurse

```
--recurse
```

Recurse into subdirectories specified in podpath (default behaviour).

norecurse

```
--norecurse
```

Do not recurse into subdirectories specified in podpath.

title

```
--title=title
```

Specify the title of the resulting HTML file.

verbose

```
--verbose
```

Display progress messages.

**EXAMPLE**

```
pod2html("pod2html",
         "--podpath=lib:ext:pod:vms",
         "--podroot=/usr/src/perl",
         "--htmlroot=/perl/nmanual",
         "--libpods=perlfunc:perlguts:perlvar:perlrun:perlop",
         "--recurse",
         "--infile=foo.pod",
         "--outfile=/perl/nmanual/foo.html");
```

**AUTHOR**

Tom Christiansen, <tchrist@perl.com>.

**BUGS**

Has trouble with  etc in = commands.

**SEE ALSO**

the *perlpod* manpage

**COPYRIGHT**

This program is distributed under the Artistic License.

## NAME

Pod::Text – convert POD data to formatted ASCII text

## SYNOPSIS

```
use Pod::Text;

pod2text("perlfunc.pod");
```

Also:

```
pod2text [B<-a>] [B<->I<width>] < input.pod
```

## DESCRIPTION

Pod::Text is a module that can convert documentation in the POD format (such as can be found throughout the Perl distribution) into formatted ASCII. Termcap is optionally supported for boldface/underline, and can enabled via `$Pod::Text::termcap=1`. If termcap has not been enabled, then backspaces will be used to simulate bold and underlined text.

A separate *pod2text* program is included that is primarily a wrapper for Pod::Text.

The single function `pod2text()` can take the optional options **–a** for an alternative output format, then a **–***width* option with the max terminal width, followed by one or two arguments. The first should be the name of a file to read the pod from, or "<&STDIN" to read from STDIN. A second argument, if provided, should be a filehandle glob where output should be sent.

## AUTHOR

Tom Christiansen <*tchrist@mox.perl.com*>

## TODO

Cleanup work. The input and output locations need to be more flexible, termcap shouldn't be a global variable, and the terminal speed needs to be properly calculated.

**NAME**

POSIX − Perl interface to IEEE Std 1003.1

**SYNOPSIS**

```
use POSIX;
use POSIX qw(setsid);
use POSIX qw(:errno_h :fcntl_h);

printf "EINTR is %d\n", EINTR;

$sess_id = POSIX::setsid();

$fd = POSIX::open($path, O_CREAT|O_EXCL|O_WRONLY, 0644);
    # note: that's a filedescriptor, *NOT* a filehandle
```

**DESCRIPTION**

The POSIX module permits you to access all (or nearly all) the standard POSIX 1003.1 identifiers. Many of these identifiers have been given Perl-ish interfaces. Things which are `#defines` in C, like EINTR or O_NDELAY, are automatically exported into your namespace. All functions are only exported if you ask for them explicitly. Most likely people will prefer to use the fully-qualified function names.

This document gives a condensed list of the features available in the POSIX module. Consult your operating system's manpages for general information on most features. Consult the *perlfunc* manpage for functions which are noted as being identical to Perl's builtin functions.

The first section describes POSIX functions from the 1003.1 specification. The second section describes some classes for signal objects, TTY objects, and other miscellaneous objects. The remaining sections list various constants and macros in an organization which roughly follows IEEE Std 1003.1b-1993.

**NOTE**

The POSIX module is probably the most complex Perl module supplied with the standard distribution. It incorporates autoloading, namespace games, and dynamic loading of code that's in Perl, C, or both. It's a great source of wisdom.

**CAVEATS**

A few functions are not implemented because they are C specific. If you attempt to call these, they will print a message telling you that they aren't implemented, and suggest using the Perl equivalent should one exist. For example, trying to access the *setjmp()* call will elicit the message "*setjmp()* is C−specific: use eval {} instead".

Furthermore, some evil vendors will claim 1003.1 compliance, but in fact are not so: they will not pass the PCTS (POSIX Compliance Test Suites). For example, one vendor may not define EDEADLK, or the semantics of the errno values set by *open*(2) might not be quite right. Perl does not attempt to verify POSIX compliance. That means you can currently successfully say "use POSIX", and then later in your program you find that your vendor has been lax and there's no usable ICANON macro after all. This could be construed to be a bug.

**FUNCTIONS**

_exit This is identical to the C function `_exit()`.

abort This is identical to the C function `abort()`.

abs This is identical to Perl's builtin `abs()` function.

access Determines the accessibility of a file.

```
if( POSIX::access( "/", &POSIX::R_OK ) ){
        print "have read permission\n";
}
```

Returns `undef` on failure.

acos This is identical to the C function `acos()`.

alarm This is identical to Perl's builtin `alarm()` function.

asctime This is identical to the C function `asctime()`.

asin This is identical to the C function `asin()`.

assert Unimplemented.

atan This is identical to the C function `atan()`.

atan2 This is identical to Perl's builtin `atan2()` function.

atexit *atexit()* is C–specific: use END {} instead.

atof *atof()* is C–specific.

atoi *atoi()* is C–specific.

atol *atol()* is C–specific.

bsearch *bsearch()* not supplied.

calloc *calloc()* is C–specific.

ceil This is identical to the C function `ceil()`.

chdir This is identical to Perl's builtin `chdir()` function.

chmod This is identical to Perl's builtin `chmod()` function.

chown This is identical to Perl's builtin `chown()` function.

clearerr Use method `IO::Handle::clearerr()` instead.

clock This is identical to the C function `clock()`.

close Close the file. This uses file descriptors such as those obtained by calling `POSIX::open`.

```
$fd = POSIX::open( "foo", &POSIX::O_RDONLY );
POSIX::close( $fd );
```

Returns `undef` on failure.

closedir This is identical to Perl's builtin `closedir()` function.

cos This is identical to Perl's builtin `cos()` function.

cosh This is identical to the C function `cosh()`.

creat Create a new file. This returns a file descriptor like the ones returned by `POSIX::open`. Use `POSIX::close` to close the file.

```
$fd = POSIX::creat( "foo", 0611 );
POSIX::close( $fd );
```

ctermid Generates the path name for the controlling terminal.

```
$path = POSIX::ctermid();
```

ctime This is identical to the C function `ctime()`.

cuserid Get the character login name of the user.

```
$name = POSIX::cuserid();
```

difftime This is identical to the C function `difftime()`.

div *div()* is C–specific.

dup This is similar to the C function `dup()`.

This uses file descriptors such as those obtained by calling `POSIX::open`.

Returns `undef` on failure.

dup2 This is similar to the C function `dup2()`.

This uses file descriptors such as those obtained by calling `POSIX::open`.

Returns `undef` on failure.

errno Returns the value of errno.

```
$errno = POSIX::errno();
```

execl *execl()* is C–specific.

execle *execle()* is C–specific.

execlp *execlp()* is C–specific.

execv *execv()* is C–specific.

execve *execve()* is C–specific.

execvp *execvp()* is C–specific.

exit This is identical to Perl's builtin `exit()` function.

exp This is identical to Perl's builtin `exp()` function.

fabs This is identical to Perl's builtin `abs()` function.

fclose Use method `IO::Handle::close()` instead.

fcntl This is identical to Perl's builtin `fcntl()` function.

fdopen Use method `IO::Handle::new_from_fd()` instead.

feof Use method `IO::Handle::eof()` instead.

ferror Use method `IO::Handle::error()` instead.

fflush Use method `IO::Handle::flush()` instead.

fgetc Use method `IO::Handle::getc()` instead.

fgetpos Use method `IO::Seekable::getpos()` instead.

fgets Use method `IO::Handle::gets()` instead.

fileno Use method `IO::Handle::fileno()` instead.

floor This is identical to the C function `floor()`.

fmod This is identical to the C function `fmod()`.

fopen Use method `IO::File::open()` instead.

fork This is identical to Perl's builtin `fork()` function.

fpathconf

Retrieves the value of a configurable limit on a file or directory. This uses file descriptors such as those obtained by calling `POSIX::open`.

The following will determine the maximum length of the longest allowable pathname on the filesystem which holds `/tmp/foo`.

```
$fd = POSIX::open( "/tmp/foo", &POSIX::O_RDONLY );
$path_max = POSIX::fpathconf( $fd, &POSIX::_PC_PATH_MAX );
```

Returns `undef` on failure.

fprintf *fprintf()* is C–specific— use printf instead.

fputc *fputc()* is C–specific— use print instead.

fputs *fputs()* is C–specific— use print instead.

fread *fread()* is C–specific— use read instead.

free *free()* is C–specific.

freopen *freopen()* is C–specific— use open instead.

frexp Return the mantissa and exponent of a floating-point number.

```
($mantissa, $exponent) = POSIX::frexp( 3.14 );
```

fscanf *fscanf()* is C–specific— use <> and regular expressions instead.

fseek Use method `IO::Seekable::seek()` instead.

fsetpos Use method `IO::Seekable::setpos()` instead.

fstat Get file status. This uses file descriptors such as those obtained by calling `POSIX::open`. The data returned is identical to the data from Perl's builtin `stat` function.

```
$fd = POSIX::open( "foo", &POSIX::O_RDONLY );
@stats = POSIX::fstat( $fd );
```

ftell Use method `IO::Seekable::tell()` instead.

fwrite *fwrite()* is C–specific— use print instead.

getc This is identical to Perl's builtin `getc()` function.

getchar Returns one character from STDIN.

getcwd Returns the name of the current working directory.

getegid Returns the effective group id.

getenv Returns the value of the specified enironment variable.

geteuid Returns the effective user id.

getgid Returns the user's real group id.

getgrgid This is identical to Perl's builtin `getgrgid()` function.

getgrnam
This is identical to Perl's builtin `getgrnam()` function.

getgroups
Returns the ids of the user's supplementary groups.

getlogin This is identical to Perl's builtin `getlogin()` function.

getpgrp This is identical to Perl's builtin `getpgrp()` function.

getpid Returns the process's id.

getppid This is identical to Perl's builtin `getppid()` function.

getpwnam
This is identical to Perl's builtin `getpwnam()` function.

getpwuid
This is identical to Perl's builtin `getpwuid()` function.

gets Returns one line from STDIN.

getuid Returns the user's id.

gmtime This is identical to Perl's builtin `gmtime()` function.

isalnum This is identical to the C function, except that it can apply to a single character or to a whole string.

isalpha This is identical to the C function, except that it can apply to a single character or to a whole string.

isatty Returns a boolean indicating whether the specified filehandle is connected to a tty.

iscntrl This is identical to the C function, except that it can apply to a single character or to a whole string.

isdigit This is identical to the C function, except that it can apply to a single character or to a whole string.

isgraph This is identical to the C function, except that it can apply to a single character or to a whole string.

islower This is identical to the C function, except that it can apply to a single character or to a whole string.

isprint This is identical to the C function, except that it can apply to a single character or to a whole string.

ispunct This is identical to the C function, except that it can apply to a single character or to a whole string.

isspace This is identical to the C function, except that it can apply to a single character or to a whole string.

isupper This is identical to the C function, except that it can apply to a single character or to a whole string.

isxdigit This is identical to the C function, except that it can apply to a single character or to a whole string.

kill This is identical to Perl's builtin `kill()` function.

labs *labs()* is C−specific, use abs instead.

ldexp This is identical to the C function `ldexp()`.

ldiv *ldiv()* is C−specific, use / and int instead.

link This is identical to Perl's builtin `link()` function.

localeconv
Get numeric formatting information. Returns a reference to a hash containing the current locale formatting values.

The database for the **de** (Deutsch or German) locale.

```
$loc = POSIX::setlocale( &POSIX::LC_ALL, "de" );
print "Locale = $loc\n";
$lconv = POSIX::localeconv();
print "decimal_point    = ", $lconv->{decimal_point},   "\n";
print "thousands_sep    = ", $lconv->{thousands_sep},   "\n";
print "grouping = ", $lconv->{grouping},        "\n";
print "int_curr_symbol  = ", $lconv->{int_curr_symbol}, "\n";
print "currency_symbol  = ", $lconv->{currency_symbol}, "\n";
print "mon_decimal_point = ", $lconv->{mon_decimal_point}, "\n";
print "mon_thousands_sep = ", $lconv->{mon_thousands_sep}, "\n";
print "mon_grouping     = ", $lconv->{mon_grouping},    "\n";
print "positive_sign    = ", $lconv->{positive_sign},   "\n";
print "negative_sign    = ", $lconv->{negative_sign},   "\n";
print "int_frac_digits  = ", $lconv->{int_frac_digits}, "\n";
print "frac_digits      = ", $lconv->{frac_digits},     "\n";
print "p_cs_precedes    = ", $lconv->{p_cs_precedes},   "\n";
print "p_sep_by_space   = ", $lconv->{p_sep_by_space},  "\n";
print "n_cs_precedes    = ", $lconv->{n_cs_precedes},   "\n";
print "n_sep_by_space   = ", $lconv->{n_sep_by_space},  "\n";
print "p_sign_posn      = ", $lconv->{p_sign_posn},     "\n";
print "n_sign_posn      = ", $lconv->{n_sign_posn},     "\n";
```

localtime
: This is identical to Perl's builtin `localtime()` function.

log
: This is identical to Perl's builtin `log()` function.

log10
: This is identical to the C function `log10()`.

longjmp
: *longjmp()* is C–specific: use die instead.

lseek
: Move the file's read/write position. This uses file descriptors such as those obtained by calling `POSIX::open`.

```
$fd = POSIX::open( "foo", &POSIX::O_RDONLY );
$off_t = POSIX::lseek( $fd, 0, &POSIX::SEEK_SET );
```

Returns `undef` on failure.

malloc
: *malloc()* is C–specific.

mblen
: This is identical to the C function `mblen()`.

mbstowcs
: This is identical to the C function `mbstowcs()`.

mbtowc
: This is identical to the C function `mbtowc()`.

memchr
: *memchr()* is C–specific, use *index()* instead.

memcmp
: *memcmp()* is C–specific, use eq instead.

memcpy
: *memcpy()* is C–specific, use = instead.

memmove
: *memmove()* is C–specific, use = instead.

memset
: *memset()* is C–specific, use x instead.

mkdir This is identical to Perl's builtin `mkdir()` function.

mkfifo This is similar to the C function `mkfifo()`.

Returns `undef` on failure.

mktime Convert date/time info to a calendar time.

Synopsis:

```
mktime(sec, min, hour, mday, mon, year, wday = 0, yday = 0, isdst = 0)
```

The month (`mon`), weekday (`wday`), and yearday (`yday`) begin at zero. I.e. January is 0, not 1; Sunday is 0, not 1; January 1st is 0, not 1. The year (`year`) is given in years since 1900. I.e. The year 1995 is 95; the year 2001 is 101. Consult your system's `mktime()` manpage for details about these and the other arguments.

Calendar time for December 12, 1995, at 10:30 am.

```
$time_t = POSIX::mktime( 0, 30, 10, 12, 11, 95 );
print "Date = ", POSIX::ctime($time_t);
```

Returns `undef` on failure.

modf Return the integral and fractional parts of a floating-point number.

```
($fractional, $integral) = POSIX::modf( 3.14 );
```

nice This is similar to the C function `nice()`.

Returns `undef` on failure.

offsetof *offsetof()* is C−specific.

open Open a file for reading for writing. This returns file descriptors, not Perl filehandles. Use `POSIX::close` to close the file.

Open a file read-only with mode 0666.

```
$fd = POSIX::open( "foo" );
```

Open a file for read and write.

```
$fd = POSIX::open( "foo", &POSIX::O_RDWR );
```

Open a file for write, with truncation.

```
$fd = POSIX::open( "foo", &POSIX::O_WRONLY | &POSIX::O_TRUNC );
```

Create a new file with mode 0640. Set up the file for writing.

```
$fd = POSIX::open( "foo", &POSIX::O_CREAT | &POSIX::O_WRONLY, 0640 );
```

Returns `undef` on failure.

opendir Open a directory for reading.

```
$dir = POSIX::opendir( "/tmp" );
@files = POSIX::readdir( $dir );
POSIX::closedir( $dir );
```

Returns `undef` on failure.

pathconf Retrieves the value of a configurable limit on a file or directory.

The following will determine the maximum length of the longest allowable pathname on the filesystem which holds `/tmp`.

```
$path_max = POSIX::pathconf( "/tmp", &POSIX::_PC_PATH_MAX );
```

Returns `undef` on failure.

pause This is similar to the C function `pause()`.

Returns `undef` on failure.

perror This is identical to the C function `perror()`.

pipe Create an interprocess channel. This returns file descriptors like those returned by `POSIX::open`.

```
($fd0, $fd1) = POSIX::pipe();
POSIX::write( $fd0, "hello", 5 );
POSIX::read( $fd1, $buf, 5 );
```

pow Computes `$x` raised to the power `$exponent`.

```
$ret = POSIX::pow( $x, $exponent );
```

printf Prints the specified arguments to STDOUT.

putc *putc()* is C–specific— use print instead.

putchar *putchar()* is C–specific— use print instead.

puts *puts()* is C–specific— use print instead.

qsort *qsort()* is C–specific, use sort instead.

raise Sends the specified signal to the current process.

rand *rand()* is non-portable, use Perl's rand instead.

read Read from a file. This uses file descriptors such as those obtained by calling `POSIX::open`. If the buffer `$buf` is not large enough for the read then Perl will extend it to make room for the request.

```
$fd = POSIX::open( "foo", &POSIX::O_RDONLY );
$bytes = POSIX::read( $fd, $buf, 3 );
```

Returns `undef` on failure.

readdir This is identical to Perl's builtin `readdir()` function.

realloc *realloc()* is C–specific.

remove This is identical to Perl's builtin `unlink()` function.

rename This is identical to Perl's builtin `rename()` function.

rewind Seeks to the beginning of the file.

rewinddir
This is identical to Perl's builtin `rewinddir()` function.

rmdir This is identical to Perl's builtin `rmdir()` function.

scanf *scanf()* is C–specific— use <> and regular expressions instead.

setgid Sets the real group id for this process.

setjmp *setjmp()* is C–specific: use eval {} instead.

setlocale Modifies and queries program's locale.

The following will set the traditional UNIX system locale behavior (the second argument `"C"`).

```
$loc = POSIX::setlocale( &POSIX::LC_ALL, "C" );
```

The following will query (the missing second argument) the current LC_CTYPE category.

```
$loc = POSIX::setlocale( &POSIX::LC_CTYPE);
```

The following will set the LC_CTYPE behaviour according to the locale environment variables (the second argument `""`). Please see your systems the *setlocale(3)* manpage documentation for the locale environment variables' meaning or consult the *perllocale* manpage.

```
$loc = POSIX::setlocale( &POSIX::LC_CTYPE, "");
```

The following will set the LC_COLLATE behaviour to Argentinian Spanish. **NOTE**: The naming and availability of locales depends on your operating system. Please consult the *perllocale* manpage for how to find out which locales are available in your system.

```
$loc = POSIX::setlocale( &POSIX::LC_ALL, "es_AR.ISO8859-1" );
```

setpgid This is similar to the C function `setpgid()`.

Returns `undef` on failure.

setsid This is identical to the C function `setsid()`.

setuid Sets the real user id for this process.

sigaction

Detailed signal management. This uses `POSIX::SigAction` objects for the `action` and `oldaction` arguments. Consult your system's `sigaction` manpage for details.

Synopsis:

```
sigaction(sig, action, oldaction = 0)
```

Returns `undef` on failure.

siglongjmp

*siglongjmp()* is C–specific: use die instead.

sigpending

Examine signals that are blocked and pending. This uses `POSIX::SigSet` objects for the `sigset` argument. Consult your system's `sigpending` manpage for details.

Synopsis:

```
sigpending(sigset)
```

Returns `undef` on failure.

sigprocmask

Change and/or examine calling process's signal mask. This uses `POSIX::SigSet` objects for the `sigset` and `oldsigset` arguments. Consult your system's `sigprocmask` manpage for details.

Synopsis:

```
sigprocmask(how, sigset, oldsigset = 0)
```

Returns `undef` on failure.

sigsetjmp
*sigsetjmp()* is C–specific: use eval {} instead.

sigsuspend
Install a signal mask and suspend process until signal arrives. This uses `POSIX::SigSet` objects for the `signal_mask` argument. Consult your system's `sigsuspend` manpage for details.

Synopsis:

```
sigsuspend(signal_mask)
```

Returns `undef` on failure.

sin This is identical to Perl's builtin `sin()` function.

sinh This is identical to the C function `sinh()`.

sleep This is identical to Perl's builtin `sleep()` function.

sprintf This is identical to Perl's builtin `sprintf()` function.

sqrt This is identical to Perl's builtin `sqrt()` function.

srand *srand()*.

sscanf *sscanf()* is C–specific—use regular expressions instead.

stat This is identical to Perl's builtin `stat()` function.

strcat *strcat()* is C–specific, use .= instead.

strchr *strchr()* is C–specific, use *index()* instead.

strcmp *strcmp()* is C–specific, use eq instead.

strcoll This is identical to the C function `strcoll()`.

strcpy *strcpy()* is C–specific, use = instead.

strcspn *strcspn()* is C–specific, use regular expressions instead.

strerror Returns the error string for the specified errno.

strftime Convert date and time information to string. Returns the string.

Synopsis:

```
strftime(fmt, sec, min, hour, mday, mon, year, wday = 0, yday = 0, isc
```

The month (`mon`), weekday (`wday`), and yearday (`yday`) begin at zero. I.e. January is 0, not 1; Sunday is 0, not 1; January 1st is 0, not 1. The year (`year`) is given in years since 1900. I.e. The year 1995 is 95; the year 2001 is 101. Consult your system's `strftime()` manpage for details about these and the other arguments.

The string for Tuesday, December 12, 1995.

```
$str = POSIX::strftime( "%A, %B %d, %Y", 0, 0, 0, 12, 11, 95, 2 );
print "$str\n";
```

strlen *strlen()* is C–specific, use length instead.

strncat *strncat()* is C−specific, use .= instead.

strncmp *strncmp()* is C−specific, use eq instead.

strncpy *strncpy()* is C−specific, use = instead.

stroul *stroul()* is C−specific.

strpbrk *strpbrk()* is C−specific.

strrchr *strrchr()* is C−specific, use *rindex()* instead.

strspn *strspn()* is C−specific.

strstr This is identical to Perl's builtin `index()` function.

strtod String to double translation. Returns the parsed number and the number of characters in the unparsed portion of the string. Truly POSIX−compliant systems set $! ($ERRNO) to indicate a translation error, so clear $! before calling strtod. However, non-POSIX systems may not check for overflow, and therefore will never set $!.

strtod should respect any POSIX *setlocale()* settings.

To parse a string `$str` as a floating point number use

```
$! = 0;
($num, $n_unparsed) = POSIX::strtod($str);
```

The second returned item and $! can be used to check for valid input:

```
if (($str eq '') || ($n_unparsed != 0) || !$!) {
    die "Non-numeric input $str" . $! ? ": $!\n" : "\n";
}
```

When called in a scalar context strtod returns the parsed number.

strtok *strtok()* is C−specific.

strtol String to (long) integer translation. Returns the parsed number and the number of characters in the unparsed portion of the string. Truly POSIX−compliant systems set $! ($ERRNO) to indicate a translation error, so clear $! before calling strtol. However, non-POSIX systems may not check for overflow, and therefore will never set $!.

strtol should respect any POSIX *setlocale()* settings.

To parse a string `$str` as a number in some base `$base` use

```
$! = 0;
($num, $n_unparsed) = POSIX::strtol($str, $base);
```

The base should be zero or between 2 and 36, inclusive. When the base is zero or omitted strtol will use the string itself to determine the base: a leading "0x" or "0X" means hexadecimal; a leading "0" means octal; any other leading characters mean decimal. Thus, "1234" is parsed as a decimal number, "01234" as an octal number, and "0x1234" as a hexadecimal number.

The second returned item and $! can be used to check for valid input:

```
if (($str eq '') || ($n_unparsed != 0) || !$!) {
    die "Non-numeric input $str" . $! ? ": $!\n" : "\n";
}
```

When called in a scalar context strtol returns the parsed number.

strtoul String to unsigned (long) integer translation. strtoul is identical to strtol except that strtoul only parses unsigned integers. See *strtol* for details.

Note: Some vendors supply strtod and strtol but not strtoul. Other vendors that do suply strtoul parse "−1" as a valid value.

strxfrm String transformation. Returns the transformed string.

```
$dst = POSIX::strxfrm( $src );
```

sysconf Retrieves values of system configurable variables.

The following will get the machine's clock speed.

```
$clock_ticks = POSIX::sysconf( &POSIX::_SC_CLK_TCK );
```

Returns `undef` on failure.

system This is identical to Perl's builtin `system()` function.

tan This is identical to the C function `tan()`.

tanh This is identical to the C function `tanh()`.

tcdrain This is similar to the C function `tcdrain()`.

Returns `undef` on failure.

tcflow This is similar to the C function `tcflow()`.

Returns `undef` on failure.

tcflush This is similar to the C function `tcflush()`.

Returns `undef` on failure.

tcgetpgrp
This is identical to the C function `tcgetpgrp()`.

tcsendbreak
This is similar to the C function `tcsendbreak()`.

Returns `undef` on failure.

tcsetpgrp
This is similar to the C function `tcsetpgrp()`.

Returns `undef` on failure.

time This is identical to Perl's builtin `time()` function.

times The *times()* function returns elapsed realtime since some point in the past (such as system startup), user and system times for this process, and user and system times used by child processes. All times are returned in clock ticks.

```
($realtime, $user, $system, $cuser, $csystem) = POSIX::times();
```

Note: Perl's builtin `times()` function returns four values, measured in seconds.

tmpfile Use method `IO::File::new_tmpfile()` instead.

tmpnam Returns a name for a temporary file.

```
$tmpfile = POSIX::tmpnam();
```

tolower This is identical to Perl's builtin `lc()` function.

toupper This is identical to Perl's builtin `uc()` function.

ttyname This is identical to the C function `ttyname()`.

tzname Retrieves the time conversion information from the `tzname` variable.

```
POSIX::tzset();
($std, $dst) = POSIX::tzname();
```

tzset This is identical to the C function `tzset()`.

umask This is identical to Perl's builtin `umask()` function.

uname Get name of current operating system.

```
($sysname, $nodename, $release, $version, $machine ) = POSIX::uname();
```

ungetc Use method `IO::Handle::ungetc()` instead.

unlink This is identical to Perl's builtin `unlink()` function.

utime This is identical to Perl's builtin `utime()` function.

vfprintf *vfprintf()* is C–specific.

vprintf *vprintf()* is C–specific.

vsprintf *vsprintf()* is C–specific.

wait This is identical to Perl's builtin `wait()` function.

waitpid Wait for a child process to change state. This is identical to Perl's builtin `waitpid()` function.

```
$pid = POSIX::waitpid( -1, &POSIX::WNOHANG );
print "status = ", ($? / 256), "\n";
```

wcstombs
This is identical to the C function `wcstombs()`.

wctomb This is identical to the C function `wctomb()`.

write Write to a file. This uses file descriptors such as those obtained by calling `POSIX::open`.

```
$fd = POSIX::open( "foo", &POSIX::O_WRONLY );
$buf = "hello";
$bytes = POSIX::write( $b, $buf, 5 );
```

Returns `undef` on failure.

## CLASSES

### POSIX::SigAction

new Creates a new `POSIX::SigAction` object which corresponds to the C `struct sigaction`. This object will be destroyed automatically when it is no longer needed. The first parameter is the fully-qualified name of a sub which is a signal-handler. The second parameter is a `POSIX::SigSet` object, it defaults to the empty set. The third parameter contains the `sa_flags`, it defaults to 0.

```
$sigset = POSIX::SigSet->new(SIGINT, SIGQUIT);
$sigaction = POSIX::SigAction->new( 'main::handler', $sigset, &POSIX:
```

This `POSIX::SigAction` object should be used with the `POSIX::sigaction()` function.

**POSIX::SigSet**

new
: Create a new SigSet object. This object will be destroyed automatically when it is no longer needed. Arguments may be supplied to initialize the set.

Create an empty set.

```
$sigset = POSIX::SigSet->new;
```

Create a set with SIGUSR1.

```
$sigset = POSIX::SigSet->new( &POSIX::SIGUSR1 );
```

addset
: Add a signal to a SigSet object.

```
$sigset->addset( &POSIX::SIGUSR2 );
```

Returns `undef` on failure.

delset
: Remove a signal from the SigSet object.

```
$sigset->delset( &POSIX::SIGUSR2 );
```

Returns `undef` on failure.

emptyset
: Initialize the SigSet object to be empty.

```
$sigset->emptyset();
```

Returns `undef` on failure.

fillset
: Initialize the SigSet object to include all signals.

```
$sigset->fillset();
```

Returns `undef` on failure.

ismember
: Tests the SigSet object to see if it contains a specific signal.

```
if( $sigset->ismember( &POSIX::SIGUSR1 ) ){
        print "contains SIGUSR1\n";
}
```

**POSIX::Termios**

new
: Create a new Termios object. This object will be destroyed automatically when it is no longer needed. A Termios object corresponds to the termios C struct. *new()* mallocs a new one, *getattr()* fills it from a file descriptor, and *setattr()* sets a file descriptor's parameters to match Termios' contents.

```
$termios = POSIX::Termios->new;
```

getattr
: Get terminal control attributes.

Obtain the attributes for stdin.

```
$termios->getattr()
```

Obtain the attributes for stdout.

```
$termios->getattr( 1 )
```

Returns `undef` on failure.

getcc Retrieve a value from the c_cc field of a termios object. The c_cc field is an array so an index must be specified.

```
$c_cc[1] = $termios->getcc(1);
```

getcflag Retrieve the c_cflag field of a termios object.

```
$c_cflag = $termios->getcflag;
```

getiflag Retrieve the c_iflag field of a termios object.

```
$c_iflag = $termios->getiflag;
```

getispeed
Retrieve the input baud rate.

```
$ispeed = $termios->getispeed;
```

getlflag Retrieve the c_lflag field of a termios object.

```
$c_lflag = $termios->getlflag;
```

getoflag Retrieve the c_oflag field of a termios object.

```
$c_oflag = $termios->getoflag;
```

getospeed
Retrieve the output baud rate.

```
$ospeed = $termios->getospeed;
```

setattr Set terminal control attributes.

Set attributes immediately for stdout.

```
$termios->setattr( 1, &POSIX::TCSANOW );
```

Returns `undef` on failure.

setcc Set a value in the c_cc field of a termios object. The c_cc field is an array so an index must be specified.

```
$termios->setcc( &POSIX::VEOF, 1 );
```

setcflag Set the c_cflag field of a termios object.

```
$termios->setcflag( $c_cflag | &POSIX::CLOCAL );
```

setiflag Set the c_iflag field of a termios object.

```
$termios->setiflag( $c_iflag | &POSIX::BRKINT );
```

setispeed
: Set the input baud rate.

```
$termios->setispeed( &POSIX::B9600 );
```

Returns `undef` on failure.

setlflag
: Set the c_lflag field of a termios object.

```
$termios->setlflag( $c_lflag | &POSIX::ECHO );
```

setoflag
: Set the c_oflag field of a termios object.

```
$termios->setoflag( $c_oflag | &POSIX::OPOST );
```

setospeed
: Set the output baud rate.

```
$termios->setospeed( &POSIX::B9600 );
```

Returns `undef` on failure.

Baud rate values
: B38400 B75 B200 B134 B300 B1800 B150 B0 B19200 B1200 B9600 B600 B4800 B50 B2400 B110

Terminal interface values
: TCSADRAIN TCSANOW TCOON TCIOFLUSH TCOFLUSH TCION TCIFLUSH TCSAFLUSH TCIOFF TCOOFF

c_cc field values
: VEOF VEOL VERASE VINTR VKILL VQUIT VSUSP VSTART VSTOP VMIN VTIME NCCS

c_cflag field values
: CLOCAL CREAD CSIZE CS5 CS6 CS7 CS8 CSTOPB HUPCL PARENB PARODD

c_iflag field values
: BRKINT ICRNL IGNBRK IGNCR IGNPAR INLCR INPCK ISTRIP IXOFF IXON PARMRK

c_lflag field values
: ECHO ECHOE ECHOK ECHONL ICANON IEXTEN ISIG NOFLSH TOSTOP

c_oflag field values
: OPOST

**PATHNAME CONSTANTS**

Constants
: _PC_CHOWN_RESTRICTED _PC_LINK_MAX _PC_MAX_CANON _PC_MAX_INPUT _PC_NAME_MAX _PC_NO_TRUNC _PC_PATH_MAX _PC_PIPE_BUF _PC_VDISABLE

**POSIX CONSTANTS**

Constants
: _POSIX_ARG_MAX _POSIX_CHILD_MAX _POSIX_CHOWN_RESTRICTED _POSIX_JOB_CONTROL _POSIX_LINK_MAX _POSIX_MAX_CANON _POSIX_MAX_INPUT _POSIX_NAME_MAX _POSIX_NGROUPS_MAX _POSIX_NO_TRUNC _POSIX_OPEN_MAX _POSIX_PATH_MAX _POSIX_PIPE_BUF _POSIX_SAVED_IDS _POSIX_SSIZE_MAX _POSIX_STREAM_MAX _POSIX_TZNAME_MAX _POSIX_VDISABLE _POSIX_VERSION

**SYSTEM CONFIGURATION**

Constants
: _SC_ARG_MAX _SC_CHILD_MAX _SC_CLK_TCK _SC_JOB_CONTROL _SC_NGROUPS_MAX _SC_OPEN_MAX _SC_SAVED_IDS _SC_STREAM_MAX _SC_TZNAME_MAX _SC_VERSION

**ERRNO**

Constants
: E2BIG EACCES EADDRINUSE EADDRNOTAVAIL EAFNOSUPPORT EAGAIN EALREADY EBADF EBUSY ECHILD ECONNABORTED ECONNREFUSED ECONNRESET EDEADLK EDESTADDRREQ EDOM EDQUOT EEXIST EFAULT EFBIG EHOSTDOWN EHOSTUNREACH EINPROGRESS EINTR EINVAL EIO EISCONN EISDIR ELOOP EMFILE EMLINK EMSGSIZE ENAMETOOLONG ENETDOWN ENETRESET ENETUNREACH ENFILE ENOBUFS ENODEV ENOENT ENOEXEC ENOLCK ENOMEM ENOPROTOOPT ENOSPC ENOSYS ENOTBLK ENOTCONN ENOTDIR ENOTEMPTY ENOTSOCK ENOTTY ENXIO EOPNOTSUPP EPERM EPFNOSUPPORT EPIPE EPROCLIM EPROTONOSUPPORT EPROTOTYPE ERANGE EREMOTE ERESTART EROFS ESHUTDOWN ESOCKTNOSUPPORT ESPIPE ESRCH ESTALE ETIMEDOUT ETOOMANYREFS ETXTBSY EUSERS EWOULDBLOCK EXDEV

**FCNTL**

Constants
: FD_CLOEXEC F_DUPFD F_GETFD F_GETFL F_GETLK F_OK F_RDLCK F_SETFD F_SETFL F_SETLK F_SETLKW F_UNLCK F_WRLCK O_ACCMODE O_APPEND O_CREAT O_EXCL O_NOCTTY O_NONBLOCK O_RDONLY O_RDWR O_TRUNC O_WRONLY

**FLOAT**

Constants
: DBL_DIG DBL_EPSILON DBL_MANT_DIG DBL_MAX DBL_MAX_10_EXP DBL_MAX_EXP DBL_MIN DBL_MIN_10_EXP DBL_MIN_EXP FLT_DIG FLT_EPSILON FLT_MANT_DIG FLT_MAX FLT_MAX_10_EXP FLT_MAX_EXP FLT_MIN FLT_MIN_10_EXP FLT_MIN_EXP FLT_RADIX FLT_ROUNDS LDBL_DIG LDBL_EPSILON LDBL_MANT_DIG LDBL_MAX LDBL_MAX_10_EXP LDBL_MAX_EXP LDBL_MIN LDBL_MIN_10_EXP LDBL_MIN_EXP

**LIMITS**

Constants
: ARG_MAX CHAR_BIT CHAR_MAX CHAR_MIN CHILD_MAX INT_MAX INT_MIN LINK_MAX LONG_MAX LONG_MIN MAX_CANON MAX_INPUT MB_LEN_MAX NAME_MAX NGROUPS_MAX OPEN_MAX PATH_MAX PIPE_BUF SCHAR_MAX SCHAR_MIN SHRT_MAX SHRT_MIN SSIZE_MAX STREAM_MAX TZNAME_MAX UCHAR_MAX UINT_MAX ULONG_MAX USHRT_MAX

**LOCALE**

Constants
: LC_ALL LC_COLLATE LC_CTYPE LC_MONETARY LC_NUMERIC LC_TIME

**MATH**

Constants
: HUGE_VAL

**SIGNAL**

Constants
: SA_NOCLDSTOP SA_NOCLDWAIT SA_NODEFER SA_ONSTACK SA_RESETHAND SA_RESTART SA_SIGINFO SIGABRT SIGALRM SIGCHLD SIGCONT SIGFPE SIGHUP SIGILL SIGINT SIGKILL SIGPIPE SIGQUIT SIGSEGV SIGSTOP SIGTERM SIGTSTP SIGTTIN SIGTTOU SIGUSR1 SIGUSR2 SIG_BLOCK SIG_DFL SIG_ERR SIG_IGN SIG_SETMASK SIG_UNBLOCK

**STAT**

Constants
S_IRGRP S_IROTH S_IRUSR S_IRWXG S_IRWXO S_IRWXU S_ISGID S_ISUID S_IWGRP S_IWOTH S_IWUSR S_IXGRP S_IXOTH S_IXUSR

Macros S_ISBLK S_ISCHR S_ISDIR S_ISFIFO S_ISREG

**STDLIB**

Constants
EXIT_FAILURE EXIT_SUCCESS MB_CUR_MAX RAND_MAX

**STDIO**

Constants
BUFSIZ EOF FILENAME_MAX L_ctermid L_cuserid L_tmpname TMP_MAX

**TIME**

Constants
CLK_TCK CLOCKS_PER_SEC

**UNISTD**

Constants
R_OK SEEK_CUR SEEK_END SEEK_SET STDIN_FILENO STDOUT_FILENO STRERR_FILENO W_OK X_OK

**WAIT**

Constants
WNOHANG WUNTRACED

Macros WIFEXITED WEXITSTATUS WIFSIGNALED WTERMSIG WIFSTOPPED WSTOPSIG

**CREATION**

This document generated by ./mkposixman.PL version 19960129.

## NAME

Safe − Compile and execute code in restricted compartments

## SYNOPSIS

```
use Safe;

$compartment = new Safe;

$compartment->permit(qw(time sort :browse));

$result = $compartment->reval($unsafe_code);
```

## DESCRIPTION

The Safe extension module allows the creation of compartments in which perl code can be evaluated. Each compartment has

a new namespace

The ‘‘root’’ of the namespace (i.e. ‘‘main::") is changed to a different package and code evaluated in the compartment cannot refer to variables outside this namespace, even with run-time glob lookups and other tricks.

Code which is compiled outside the compartment can choose to place variables into (or *share* variables with) the compartment's namespace and only that data will be visible to code evaluated in the compartment.

By default, the only variables shared with compartments are the ‘‘underscore’’ variables `$_` and `@_` (and, technically, the less frequently used `%_`, the _ filehandle and so on). This is because otherwise perl operators which default to `$_` will not work and neither will the assignment of arguments to `@_` on subroutine entry.

an operator mask

Each compartment has an associated ‘‘operator mask’’. Recall that perl code is compiled into an internal format before execution. Evaluating perl code (e.g. via ‘‘eval’’ or ‘‘do ‘file’") causes the code to be compiled into an internal format and then, provided there was no error in the compilation, executed. Code evaulated in a compartment compiles subject to the compartment's operator mask. Attempting to evaulate code in a compartment which contains a masked operator will cause the compilation to fail with an error. The code will not be executed.

The default operator mask for a newly created compartment is the ‘:default’ optag.

It is important that you read the *Opcode*(3) module documentation for more information, especially for detailed definitions of opnames, optags and opsets.

Since it is only at the compilation stage that the operator mask applies, controlled access to potentially unsafe operations can be achieved by having a handle to a wrapper subroutine (written outside the compartment) placed into the compartment. For example,

```
$cpt = new Safe;
sub wrapper {
    # vet arguments and perform potentially unsafe operations
}
$cpt->share('&wrapper');
```

## WARNING

The authors make **no warranty**, implied or otherwise, about the suitability of this software for safety or security purposes.

The authors shall not in any case be liable for special, incidental, consequential, indirect or other similar damages arising from the use of this software.

Your mileage will vary. If in any doubt **do not use it**.

**RECENT CHANGES**

The interface to the Safe module has changed quite dramatically since version 1 (as supplied with Perl5.002). Study these pages carefully if you have code written to use Safe version 1 because you will need to makes changes.

## Methods in class Safe

To create a new compartment, use

```
$cpt = new Safe;
```

Optional argument is (NAMESPACE), where NAMESPACE is the root namespace to use for the compartment (defaults to ''Safe::Root0'', incremented for each new compartment).

Note that version 1.00 of the Safe module supported a second optional parameter, MASK. That functionality has been withdrawn pending deeper consideration. Use the permit and deny methods described below.

The following methods can then be used on the compartment object returned by the above constructor. The object argument is implicit in each case.

permit (OP, ...)
: Permit the listed operators to be used when compiling code in the compartment (in *addition* to any operators already permitted).

permit_only (OP, ...)
: Permit *only* the listed operators to be used when compiling code in the compartment (*no* other operators are permitted).

deny (OP, ...)
: Deny the listed operators from being used when compiling code in the compartment (other operators may still be permitted).

deny_only (OP, ...)
: Deny *only* the listed operators from being used when compiling code in the compartment (*all* other operators will be permitted).

trap (OP, ...)

untrap (OP, ...)
: The trap and untrap methods are synonyms for deny and permit respectfully.

share (NAME, ...)
: This shares the *variable* (s) in the argument list with the compartment. This is almost identical to exporting variables using the the *Exporter(3)* manpage module.

  Each NAME must be the **name** of a variable, typically with the leading type identifier included. A bareword is treated as a function name.

  Examples of legal names are '$foo' for a scalar, '@foo' for an array, '%foo' for a hash, '&foo' or 'foo' for a subroutine and '*foo' for a glob (i.e. all symbol table entries associated with ''foo'', including scalar, array, hash, sub and filehandle).

  Each NAME is assumed to be in the calling package. See share_from for an alternative method (which share uses).

share_from (PACKAGE, ARRAYREF)
: This method is similar to *share()* but allows you to explicitly name the package that symbols should be shared from. The symbol names (including type characters) are supplied as an array reference.

```
$safe->share_from('main', [ '$foo', '%bar', 'func' ]);
```

varglob (VARNAME)

This returns a glob reference for the symbol table entry of VARNAME in the package of the compartment. VARNAME must be the **name** of a variable without any leading type marker. For example,

```
$cpt = new Safe 'Root';
$Root::foo = "Hello world";
# Equivalent version which doesn't need to know $cpt's package name:
${$cpt->varglob('foo')} = "Hello world";
```

reval (STRING)

This evaluates STRING as perl code inside the compartment.

The code can only see the compartment's namespace (as returned by the **root** method). The compartment's root package appears to be the `main::` package to the code inside the compartment.

Any attempt by the code in STRING to use an operator which is not permitted by the compartment will cause an error (at run-time of the main program but at compile-time for the code in STRING). The error is of the form "%s trapped by operation mask operation...".

If an operation is trapped in this way, then the code in STRING will not be executed. If such a trapped operation occurs or any other compile-time or return error, then $@ is set to the error message, just as with an *eval()*.

If there is no error, then the method returns the value of the last expression evaluated, or a return statement may be used, just as with subroutines and **eval()**. The context (list or scalar) is determined by the caller as usual.

This behaviour differs from the beta distribution of the Safe extension where earlier versions of perl made it hard to mimic the return behaviour of the *eval()* command and the context was always scalar.

Some points to note:

If the entereval op is permitted then the code can use eval "..." to 'hide' code which might use denied ops. This is not a major problem since when the code tries to execute the eval it will fail because the opmask is still in effect. However this technique would allow clever, and possibly harmful, code to 'probe' the boundaries of what is possible.

Any string eval which is executed by code executing in a compartment, or by code called from code executing in a compartment, will be eval'd in the namespace of the compartment. This is potentially a serious problem.

Consider a function *foo()* in package pkg compiled outside a compartment but shared with it. Assume the compartment has a root package called 'Root'. If *foo()* contains an eval statement like eval '$foo = 1' then, normally, `$pkg::foo` will be set to 1. If *foo()* is called from the compartment (by whatever means) then instead of setting `$pkg::foo`, the eval will actually set `$Root::pkg::foo`.

This can easily be demonstrated by using a module, such as the Socket module, which uses eval "..." as part of an AUTOLOAD function. You can 'use' the module outside the compartment and share an (autoloaded) function with the compartment. If an autoload is triggered by code in the compartment, or by any code anywhere that is called by any means from the compartment, then the eval in the Socket module's AUTOLOAD function happens in the namespace of the compartment. Any variables created or used by the eval'd code are now under the control of the code in the compartment.

A similar effect applies to *all* runtime symbol lookups in code called from a compartment but not compiled within it.

rdo (FILENAME)
: This evaluates the contents of file FILENAME inside the compartment. See above documentation on the **reval** method for further details.

root (NAMESPACE)
: This method returns the name of the package that is the root of the compartment's namespace.

  Note that this behaviour differs from version 1.00 of the Safe module where the root module could be used to change the namespace. That functionality has been withdrawn pending deeper consideration.

mask (MASK)
: This is a get-or-set method for the compartment's operator mask.

  With no MASK argument present, it returns the current operator mask of the compartment.

  With the MASK argument present, it sets the operator mask for the compartment (equivalent to calling the deny_only method).

### Some Safety Issues

This section is currently just an outline of some of the things code in a compartment might do (intentionally or unintentionally) which can have an effect outside the compartment.

Memory
: Consuming all (or nearly all) available memory.

CPU
: Causing infinite loops etc.

Snooping
: Copying private information out of your system. Even something as simple as your user name is of value to others. Much useful information could be gleaned from your environment variables for example.

Signals
: Causing signals (especially SIGFPE and SIGALARM) to affect your process.

  Setting up a signal handler will need to be carefully considered and controlled. What mask is in effect when a signal handler gets called? If a user can get an imported function to get an exception and call the user's signal handler, does that user's restricted mask get re-instated before the handler is called? Does an imported handler get called with its original mask or the user's one?

State Changes
: Ops such as chdir obviously effect the process as a whole and not just the code in the compartment. Ops such as rand and srand have a similar but more subtle effect.

## AUTHOR

Originally designed and implemented by Malcolm Beattie, mbeattie@sable.ox.ac.uk.

Reworked to use the Opcode module and other changes added by Tim Bunce <*Tim.Bunce@ig.co.uk*>.

## NAME

SDBM_File – Tied access to sdbm files

## SYNOPSIS

```
use SDBM_File;

tie(%h, 'SDBM_File', 'Op.dbmx', O_RDWR|O_CREAT, 0640);

untie %h;
```

## DESCRIPTION

See the `tie` entry in the *perlfunc* manpage

**NAME**

Search::Dict, look − search for key in dictionary file

**SYNOPSIS**

```
use Search::Dict;
look *FILEHANDLE, $key, $dict, $fold;
```

**DESCRIPTION**

Sets file position in FILEHANDLE to be first line greater than or equal (stringwise) to *$key*. Returns the new file position, or −1 if an error occurs.

The flags specify dictionary order and case folding:

If *$dict* is true, search by dictionary order (ignore anything but word characters and whitespace).

If *$fold* is true, ignore case.

## NAME

SelectSaver – save and restore selected file handle

## SYNOPSIS

```
use SelectSaver;

{
   my $saver = new SelectSaver(FILEHANDLE);
   # FILEHANDLE is selected
}
# previous handle is selected

{
   my $saver = new SelectSaver;
   # new handle may be selected, or not
}
# previous handle is selected
```

## DESCRIPTION

A `SelectSaver` object contains a reference to the file handle that was selected when it was created. If its `new` method gets an extra parameter, then that parameter is selected; otherwise, the selected file handle remains unchanged.

When a `SelectSaver` is destroyed, it re-selects the file handle that was selected when it was created.

**NAME**

SelfLoader − load functions only on demand

**SYNOPSIS**

```
package FOOBAR;
use SelfLoader;

... (initializing code)

__DATA__
sub {....
```

**DESCRIPTION**

This module tells its users that functions in the FOOBAR package are to be autoloaded from after the `__DATA__` token. See also the section on *Autoloading* in the *perlsub* manpage.

### The __DATA__ token

The `__DATA__` token tells the perl compiler that the perl code for compilation is finished. Everything after the `__DATA__` token is available for reading via the filehandle FOOBAR::DATA, where FOOBAR is the name of the current package when the `__DATA__` token is reached. This works just the same as `__END__` does in package 'main', but for other modules data after `__END__` is not automatically retreivable , whereas data after `__DATA__` is. The `__DATA__` token is not recognized in versions of perl prior to 5.001m.

Note that it is possible to have `__DATA__` tokens in the same package in multiple files, and that the last `__DATA__` token in a given package that is encountered by the compiler is the one accessible by the filehandle. This also applies to `__END__` and main, i.e. if the 'main' program has an `__END__`, but a module 'require'd (_not_ 'use'd) by that program has a 'package main;' declaration followed by an '`__DATA__`', then the `DATA` filehandle is set to access the data after the `__DATA__` in the module, _not_ the data after the `__END__` token in the 'main' program, since the compiler encounters the 'require'd file later.

### SelfLoader autoloading

The **SelfLoader** works by the user placing the `__DATA__` token *after* perl code which needs to be compiled and run at 'require' time, but *before* subroutine declarations that can be loaded in later − usually because they may never be called.

The **SelfLoader** will read from the FOOBAR::DATA filehandle to load in the data after `__DATA__`, and load in any subroutine when it is called. The costs are the one-time parsing of the data after `__DATA__`, and a load delay for the _first_ call of any autoloaded function. The benefits (hopefully) are a speeded up compilation phase, with no need to load functions which are never used.

The **SelfLoader** will stop reading from `__DATA__` if it encounters the `__END__` token − just as you would expect. If the `__END__` token is present, and is followed by the token DATA, then the **SelfLoader** leaves the FOOBAR::DATA filehandle open on the line after that token.

The **SelfLoader** exports the `AUTOLOAD` subroutine to the package using the **SelfLoader**, and this loads the called subroutine when it is first called.

There is no advantage to putting subroutines which will _always_ be called after the `__DATA__` token.

### Autoloading and package lexicals

A 'my `$pack_lexical`' statement makes the variable `$pack_lexical` local _only_ to the file up to the `__DATA__` token. Subroutines declared elsewhere _cannot_ see these types of variables, just as if you declared subroutines in the package but in another file, they cannot see these variables.

So specifically, autoloaded functions cannot see package lexicals (this applies to both the **SelfLoader** and the Autoloader). The `vars` pragma provides an alternative to defining package-level globals that will be visible to autoloaded routines. See the documentation on **vars** in the pragma section of the *perlmod* manpage.

### SelfLoader and AutoLoader

The **SelfLoader** can replace the AutoLoader – just change 'use AutoLoader' to 'use SelfLoader' (though note that the **SelfLoader** exports the AUTOLOAD function – but if you have your own AUTOLOAD and are using the AutoLoader too, you probably know what you're doing), and the `__END__` token to `__DATA__`. You will need perl version 5.001m or later to use this (version 5.001 with all patches up to patch m).

There is no need to inherit from the **SelfLoader**.

The **SelfLoader** works similarly to the AutoLoader, but picks up the subs from after the `__DATA__` instead of in the 'lib/auto' directory. There is a maintainance gain in not needing to run AutoSplit on the module at installation, and a runtime gain in not needing to keep opening and closing files to load subs. There is a runtime loss in needing to parse the code after the `__DATA__`. Details of the **AutoLoader** and another view of these distinctions can be found in that module's documentation.

### __DATA__, __END__, and the FOOBAR::DATA filehandle.

This section is only relevant if you want to use the `FOOBAR::DATA` together with the **SelfLoader**.

Data after the `__DATA__` token in a module is read using the FOOBAR::DATA filehandle. `__END__` can still be used to denote the end of the `__DATA__` section if followed by the token DATA – this is supported by the **SelfLoader**. The `FOOBAR::DATA` filehandle is left open if an `__END__` followed by a DATA is found, with the filehandle positioned at the start of the line after the `__END__` token. If no `__END__` token is present, or an `__END__` token with no DATA token on the same line, then the filehandle is closed.

The **SelfLoader** reads from wherever the current position of the `FOOBAR::DATA` filehandle is, until the EOF or `__END__`. This means that if you want to use that filehandle (and ONLY if you want to), you should either

1. Put all your subroutine declarations immediately after the `__DATA__` token and put your own data after those declarations, using the `__END__` token to mark the end of subroutine declarations. You must also ensure that the **SelfLoader** reads first by calling 'SelfLoader->*load_stubs()*;', or by using a function which is selfloaded;

or

2. You should read the `FOOBAR::DATA` filehandle first, leaving the handle open and positioned at the first line of subroutine declarations.

You could conceivably do both.

### Classes and inherited methods.

For modules which are not classes, this section is not relevant. This section is only relevant if you have methods which could be inherited.

A subroutine stub (or forward declaration) looks like

```
sub stub;
```

i.e. it is a subroutine declaration without the body of the subroutine. For modules which are not classes, there is no real need for stubs as far as autoloading is concerned.

For modules which ARE classes, and need to handle inherited methods, stubs are needed to ensure that the method inheritance mechanism works properly. You can load the stubs into the module at 'require' time, by adding the statement 'SelfLoader->*load_stubs()*;' to the module to do this.

The alternative is to put the stubs in before the `__DATA__` token BEFORE releasing the module, and for this purpose the `Devel::SelfStubber` module is available. However this does require the extra step of ensuring that the stubs are in the module. If this is done I strongly recommend that this is done BEFORE releasing the module – it should NOT be done at install time in general.

### Multiple packages and fully qualified subroutine names

Subroutines in multiple packages within the same file are supported – but you should note that this requires exporting the `SelfLoader::AUTOLOAD` to every package which requires it. This is done automatically by the **SelfLoader** when it first loads the subs into the cache, but you should really specify it in the initialization before the `__DATA__` by putting a 'use SelfLoader' statement in each package.

Fully qualified subroutine names are also supported. For example,

```
__DATA__
sub foo::bar {23}
package baz;
sub dob {32}
```

will all be loaded correctly by the **SelfLoader**, and the **SelfLoader** will ensure that the packages 'foo' and 'baz' correctly have the **SelfLoader** `AUTOLOAD` method when the data after `__DATA__` is first parsed.

**NAME**

Shell – run shell commands transparently within perl

**SYNOPSIS**

See below.

**DESCRIPTION**

```
Date: Thu, 22 Sep 94 16:18:16 -0700
Message-Id: <9409222318.AA17072@scalpel.netlabs.com>
To: perl5-porters@isu.edu
From: Larry Wall <lwall@scalpel.netlabs.com>
Subject: a new module I just wrote
```

Here's one that'll whack your mind a little out.

```
#!/usr/bin/perl

use Shell;

$foo = echo("howdy", "<funny>", "world");
print $foo;

$passwd = cat("</etc/passwd");
print $passwd;

sub ps;
print ps -ww;

cp("/etc/passwd", "/tmp/passwd");
```

That's maybe too gonzo. It actually exports an AUTOLOAD to the current package (and uncovered a bug in Beta 3, by the way). Maybe the usual usage should be

```
use Shell qw(echo cat ps cp);
```

Larry

**AUTHOR**

Larry Wall

## NAME

Socket, sockaddr_in, sockaddr_un, inet_aton, inet_ntoa – load the C socket.h defines and structure manipulators

## SYNOPSIS

```
use Socket;

$proto = getprotobyname('udp');
socket(Socket_Handle, PF_INET, SOCK_DGRAM, $proto);
$iaddr = gethostbyname('hishost.com');
$port = getservbyname('time', 'udp');
$sin = sockaddr_in($port, $iaddr);
send(Socket_Handle, 0, 0, $sin);

$proto = getprotobyname('tcp');
socket(Socket_Handle, PF_INET, SOCK_STREAM, $proto);
$port = getservbyname('smtp', 'tcp');
$sin = sockaddr_in($port,inet_aton("127.1"));
$sin = sockaddr_in(7,inet_aton("localhost"));
$sin = sockaddr_in(7,INADDR_LOOPBACK);
connect(Socket_Handle,$sin);

($port, $iaddr) = sockaddr_in(getpeername(Socket_Handle));
$peer_host = gethostbyaddr($iaddr, AF_INET);
$peer_addr = inet_ntoa($iaddr);

$proto = getprotobyname('tcp');
socket(Socket_Handle, PF_UNIX, SOCK_STREAM, $proto);
unlink('/tmp/usock');
$sun = sockaddr_un('/tmp/usock');
connect(Socket_Handle,$sun);
```

## DESCRIPTION

This module is just a translation of the C *socket.h* file. Unlike the old mechanism of requiring a translated *socket.ph* file, this uses the **h2xs** program (see the Perl source distribution) and your native C compiler. This means that it has a far more likely chance of getting the numbers right. This includes all of the commonly used pound-defines like AF_INET, SOCK_STREAM, etc.

Also, some common socket "newline" constants are provided: the constants `CR`, `LF`, and `CRLF`, as well as `$CR`, `$LF`, and `$CRLF`, which map to `\015`, `\012`, and `\015\012`. If you do not want to use the literal characters in your programs, then use the constants provided here. They are not exported by default, but can be imported individually, and with the `:crlf` export tag:

```
use Socket qw(:DEFAULT :crlf);
```

In addition, some structure manipulation functions are available:

inet_aton HOSTNAME
: Takes a string giving the name of a host, and translates that to the 4-byte string (structure). Takes arguments of both the 'rtfm.mit.edu' type and '18.181.0.24'. If the host name cannot be resolved, returns undef. For multi-homed hosts (hosts with more than one address), the first address found is returned.

inet_ntoa IP_ADDRESS
: Takes a four byte ip address (as returned by *inet_aton()*) and translates it into a string of the form 'd.d.d.d' where the 'd's are numbers less than 256 (the normal readable four dotted number notation for internet addresses).

INADDR_ANY

Note: does not return a number, but a packed string.

Returns the 4-byte wildcard ip address which specifies any of the hosts ip addresses. (A particular machine can have more than one ip address, each address corresponding to a particular network interface. This wildcard address allows you to bind to all of them simultaneously.) Normally equivalent to *inet_aton* ('0.0.0.0').

INADDR_BROADCAST

Note: does not return a number, but a packed string.

Returns the 4-byte 'this-lan' ip broadcast address. This can be useful for some protocols to solicit information from all servers on the same LAN cable. Normally equivalent to *inet_aton* ('255.255.255.255').

INADDR_LOOPBACK

Note – does not return a number.

Returns the 4-byte loopback address. Normally equivalent to *inet_aton* ('localhost').

INADDR_NONE

Note – does not return a number.

Returns the 4-byte 'invalid' ip address. Normally equivalent to *inet_aton* ('255.255.255.255').

sockaddr_in PORT, ADDRESS

sockaddr_in SOCKADDR_IN

In an array context, unpacks its SOCKADDR_IN argument and returns an array consisting of (PORT, ADDRESS). In a scalar context, packs its (PORT, ADDRESS) arguments as a SOCKADDR_IN and returns it. If this is confusing, use *pack_sockaddr_in()* and *unpack_sockaddr_in()* explicitly.

pack_sockaddr_in PORT, IP_ADDRESS

Takes two arguments, a port number and a 4 byte IP_ADDRESS (as returned by *inet_aton()*). Returns the sockaddr_in structure with those arguments packed in with AF_INET filled in. For internet domain sockets, this structure is normally what you need for the arguments in *bind()*, *connect()*, and *send()*, and is also returned by *getpeername()*, *getsockname()* and *recv()*.

unpack_sockaddr_in SOCKADDR_IN

Takes a sockaddr_in structure (as returned by *pack_sockaddr_in()*) and returns an array of two elements: the port and the 4-byte ip-address. Will croak if the structure does not have AF_INET in the right place.

sockaddr_un PATHNAME

sockaddr_un SOCKADDR_UN

In an array context, unpacks its SOCKADDR_UN argument and returns an array consisting of (PATHNAME). In a scalar context, packs its PATHNAME arguments as a SOCKADDR_UN and returns it. If this is confusing, use *pack_sockaddr_un()* and *unpack_sockaddr_un()* explicitly. These are only supported if your system has <*sys/un.h*>.

pack_sockaddr_un PATH

Takes one argument, a pathname. Returns the sockaddr_un structure with that path packed in with AF_UNIX filled in. For unix domain sockets, this structure is normally what you need for the arguments in *bind()*, *connect()*, and *send()*, and is also returned by *getpeername()*, *getsockname()* and *recv()*.

unpack_sockaddr_un SOCKADDR_UN

Takes a sockaddr_un structure (as returned by *pack_sockaddr_un()*) and returns the pathname. Will croak if the structure does not have AF_UNIX in the right place.

## NAME

Symbol − manipulate Perl symbols and their names

## SYNOPSIS

```
use Symbol;

$sym = gensym;
open($sym, "filename");
$_ = <$sym>;
# etc.

ungensym $sym;      # no effect

print qualify("x"), "\n";              # "Test::x"
print qualify("x", "FOO"), "\n"        # "FOO::x"
print qualify("BAR::x"), "\n";         # "BAR::x"
print qualify("BAR::x", "FOO"), "\n";  # "BAR::x"
print qualify("STDOUT", "FOO"), "\n";  # "main::STDOUT" (global)
print qualify(\*x), "\n";              # returns \*x
print qualify(\*x, "FOO"), "\n";       # returns \*x

use strict refs;
print { qualify_to_ref $fh } "foo!\n";
$ref = qualify_to_ref $name, $pkg;

use Symbol qw(delete_package);
delete_package('Foo::Bar');
print "deleted\n" unless exists $Foo::{'Bar::'};
```

## DESCRIPTION

`Symbol::gensym` creates an anonymous glob and returns a reference to it. Such a glob reference can be used as a file or directory handle.

For backward compatibility with older implementations that didn't support anonymous globs, `Symbol::ungensym` is also provided. But it doesn't do anything.

`Symbol::qualify` turns unqualified symbol names into qualified variable names (e.g. "myvar" −> "MyPackage::myvar"). If it is given a second parameter, `qualify` uses it as the default package; otherwise, it uses the package of its caller. Regardless, global variable names (e.g. "STDOUT", "ENV", "SIG") are always qualfied with "main::".

Qualification applies only to symbol names (strings). References are left unchanged under the assumption that they are glob references, which are qualified by their nature.

`Symbol::qualify_to_ref` is just like `Symbol::qualify` except that it returns a glob ref rather than a symbol name, so you can use the result even if `use strict 'refs'` is in effect.

`Symbol::delete_package` wipes out a whole package namespace. Note this routine is not exported by default— you may want to import it explicitly.

**NAME**

Sys::Hostname − Try every conceivable way to get hostname

**SYNOPSIS**

```
use Sys::Hostname;
$host = hostname;
```

**DESCRIPTION**

Attempts several methods of getting the system hostname and then caches the result. It tries `syscall(SYS_gethostname)`, ``hostname``, ``uname -n``, and the file */com/host*. If all that fails it `croak`s.

All nulls, returns, and newlines are removed from the result.

**AUTHOR**

David Sundstrom <*sunds@asictest.sc.ti.com*>

Texas Instruments

## NAME

Sys::Syslog, openlog, closelog, setlogmask, syslog – Perl interface to the UNIX *syslog* (3) calls

## SYNOPSIS

```
use Sys::Syslog;                           # all except setlogsock, or:
use Sys::Syslog qw(:DEFAULT setlogsock);  # default set, plus setlogsock

setlogsock $sock_type;
openlog $ident, $logopt, $facility;
syslog $priority, $format, @args;
$oldmask = setlogmask $mask_priority;
closelog;
```

## DESCRIPTION

Sys::Syslog is an interface to the UNIX `syslog(3)` program. Call `syslog()` with a string priority and a list of `printf()` args just like `syslog(3)`.

Syslog provides the functions:

openlog $ident, $logopt, $facility
: *$ident* is prepended to every message. *$logopt* contains zero or more of the words *pid*, *ndelay*, *cons*, *nowait*. *$facility* specifies the part of the system

syslog $priority, $format, @args
: If *$priority* permits, logs *($format, @args)* printed as by `printf(3V)`, with the addition that *%m* is replaced with `"$!"` (the latest error message).

setlogmask $mask_priority
: Sets log mask *$mask_priority* and returns the old mask.

setlogsock $sock_type (added in 5.004_02)
: Sets the socket type to be used for the next call to `openlog()` or `syslog()` and returns TRUE on success, undef on failure.

A value of 'unix' will connect to the UNIX domain socket returned by `_PATH_LOG` in *syslog.ph*. A value of 'inet' will connect to an INET socket returned by *getservbyname()*. Any other value croaks.

The default is for the INET socket to be used.

closelog
: Closes the log file.

Note that `openlog` now takes three arguments, just like `openlog(3)`.

## EXAMPLES

```
openlog($program, 'cons,pid', 'user');
syslog('info', 'this is another test');
syslog('mail|warning', 'this is a better test: %d', time);
closelog();

syslog('debug', 'this is the last test');

setlogsock('unix');
openlog("$program $$", 'ndelay', 'user');
syslog('notice', 'fooprogram: this is really done');

setlogsock('inet');
$! = 55;
syslog('info', 'problem was %m'); # %m == $! in syslog(3)
```

### DEPENDENCIES

**Sys::Syslog** needs *syslog.ph*, which can be created with `h2ph`.

### SEE ALSO

the *syslog(3)* manpage

### AUTHOR

Tom Christiansen <*tchrist@perl.com*> and Larry Wall <*larry@wall.org*>. UNIX domain sockets added by Sean Robinson <*robinson_s@sc.maricopa.edu*> with support from Tim Bunce <Tim.Bunce@ig.co.uk> and the perl5-porters mailing list.

## NAME

Term::Cap – Perl termcap interface

## SYNOPSIS

```
require Term::Cap;
$terminal = Tgetent Term::Cap { TERM => undef, OSPEED => $ospeed };
$terminal->Trequire(qw/ce ku kd/);
$terminal->Tgoto('cm', $col, $row, $FH);
$terminal->Tputs('dl', $count, $FH);
$terminal->Tpad($string, $count, $FH);
```

## DESCRIPTION

These are low-level functions to extract and use capabilities from a terminal capability (termcap) database.

The **Tgetent** function extracts the entry of the specified terminal type *TERM* (defaults to the environment variable *TERM*) from the database.

It will look in the environment for a *TERMCAP* variable. If found, and the value does not begin with a slash, and the terminal type name is the same as the environment string *TERM*, the *TERMCAP* string is used instead of reading a termcap file. If it does begin with a slash, the string is used as a path name of the termcap file to search. If *TERMCAP* does not begin with a slash and name is different from *TERM*, **Tgetent** searches the files *$HOME/.termcap*, */etc/termcap*, and */usr/share/misc/termcap*, in that order, unless the environment variable *TERMPATH* exists, in which case it specifies a list of file pathnames (separated by spaces or colons) to be searched **instead**. Whenever multiple files are searched and a tc field occurs in the requested entry, the entry it names must be found in the same file or one of the succeeding files. If there is a `:tc=...:` in the *TERMCAP* environment variable string it will continue the search in the files as above.

*OSPEED* is the terminal output bit rate (often mistakenly called the baud rate). *OSPEED* can be specified as either a POSIX termios/SYSV termio speeds (where 9600 equals 9600) or an old BSD–style speeds (where 13 equals 9600).

**Tgetent** returns a blessed object reference which the user can then use to send the control strings to the terminal using **Tputs** and **Tgoto**. It calls `croak` on failure.

**Tgoto** decodes a cursor addressing string with the given parameters.

The output strings for **Tputs** are cached for counts of 1 for performance. **Tgoto** and **Tpad** do not cache. `$self->{_xx}` is the raw termcap data and `$self->{xx}` is the cached version.

```
print $terminal->Tpad($self->{_xx}, 1);
```

**Tgoto**, **Tputs**, and **Tpad** return the string and will also output the string to `$FH` if specified.

The extracted termcap entry is available in the object as `$self->{TERMCAP}`.

## EXAMPLES

```
# Get terminal output speed
require POSIX;
my $termios = new POSIX::Termios;
$termios->getattr;
my $ospeed = $termios->getospeed;

# Old-style ioctl code to get ospeed:
#     require 'ioctl.pl';
#     ioctl(TTY,$TIOCGETP,$sgtty);
#     ($ispeed,$ospeed) = unpack('cc',$sgtty);

# allocate and initialize a terminal structure
$terminal = Tgetent Term::Cap { TERM => undef, OSPEED => $ospeed };
```

```
# require certain capabilities to be available
$terminal->Trequire(qw/ce ku kd/);

# Output Routines, if $FH is undefined these just return the string

# Tgoto does the % expansion stuff with the given args
$terminal->Tgoto('cm', $col, $row, $FH);

# Tputs doesn't do any % expansion.
$terminal->Tputs('dl', $count = 1, $FH);
```

## NAME

Term::Complete – Perl word completion module

## SYNOPSIS

```
$input = complete('prompt_string', \@completion_list);
$input = complete('prompt_string', @completion_list);
```

## DESCRIPTION

This routine provides word completion on the list of words in the array (or array ref).

The tty driver is put into raw mode using the system command `stty raw -echo` and restored using `stty -raw echo`.

The following command characters are defined:

<tab>
Attempts word completion. Cannot be changed.

^D Prints completion list. Defined by *$Term::Complete::complete*.

^U Erases the current input. Defined by *$Term::Complete::kill*.

<del>, <bs>
Erases one character. Defined by *$Term::Complete::erase1* and *$Term::Complete::erase2*.

## DIAGNOSTICS

Bell sounds when word completion fails.

## BUGS

The completion charater <tab> cannot be changed.

## AUTHOR

Wayne Thompson

## NAME

Term::ReadLine − Perl interface to various `readline` packages. If no real package is found, substitutes stubs instead of basic functions.

## SYNOPSIS

```
use Term::ReadLine;
$term = new Term::ReadLine 'Simple Perl calc';
$prompt = "Enter your arithmetic expression: ";
$OUT = $term->OUT || STDOUT;
while ( defined ($_ = $term->readline($prompt)) ) {
  $res = eval($_), "\n";
  warn $@ if $@;
  print $OUT $res, "\n" unless $@;
  $term->addhistory($_) if /\S/;
}
```

## DESCRIPTION

This package is just a front end to some other packages. At the moment this description is written, the only such package is Term-ReadLine, available on CPAN near you. The real target of this stub package is to set up a common interface to whatever Readline emerges with time.

### Minimal set of supported functions

All the supported functions should be called as methods, i.e., either as

```
$term = new Term::ReadLine 'name';
```

or as

```
$term->addhistory('row');
```

where `$term` is a return value of Term::ReadLine->Init.

`ReadLine` returns the actual package that executes the commands. Among possible values are `Term::ReadLine::Gnu`, `Term::ReadLine::Perl`, `Term::ReadLine::Stub Exporter`.

`new` returns the handle for subsequent calls to following functions. Argument is the name of the application. Optionally can be followed by two arguments for `IN` and `OUT` filehandles. These arguments should be globs.

`readline` gets an input line, *possibly* with actual `readline` support. Trailing newline is removed. Returns `undef` on `EOF`.

`addhistory`
adds the line to the history of input, from where it can be used if the actual `readline` is present.

`IN`, $`OUT` return the filehandles for input and output or `undef` if `readline` input and output cannot be used for Perl.

`MinLine` If argument is specified, it is an advice on minimal size of line to be included into history. `undef` means do not include anything into history. Returns the old value.

`findConsole`
returns an array with two strings that give most appropriate names for files for input and output using conventions `"<$in"`, `">out"`.

Attribs returns a reference to a hash which describes internal configuration of the package. Names of keys in this hash conform to standard conventions with the leading `rl_` stripped.

`Features` Returns a reference to a hash with keys being features present in current implementation. Several optional features are used in the minimal interface: `appname` should be present if the first argument to `new` is recognized, and `minline` should be present if `MinLine` method is not dummy. `autohistory` should be present if lines are put into history automatically (maybe subject to `MinLine`), and `addhistory` if `addhistory` method is not dummy.

If `Features` method reports a feature `attribs` as present, the method `Attribs` is not dummy.

**Additional supported functions**

Actually `Term::ReadLine` can use some other package, that will support reacher set of commands.

All these commands are callable via method interface and have names which conform to standard conventions with the leading `rl_` stripped.

The stub package included with the perl distribution allows some additional methods:

`tkRunning` makes Tk event loop run when waiting for user input (i.e., during `readline` method).

`ornaments` makes the command line stand out by using termcap data. The argument to `ornaments` should be 0, 1, or a string of a form `"aa,bb,cc,dd"`. Four components of this string should be names of *terminal capacities*, first two will be issued to make the prompt standout, last two to make the input line standout.

`newTTY` takes two arguments which are input filehandle and output filehandle. Switches to use these filehandles.

One can check whether the currently loaded ReadLine package supports these methods by checking for corresponding `Features`.

**EXPORTS**

None

**ENVIRONMENT**

The envrironment variable `PERL_RL` governs which ReadLine clone is loaded. If the value is false, a dummy interface is used. If the value is true, it should be tail of the name of the package to use, such as `Perl` or `Gnu`.

As a special case, if the value of this variable is space-separated, the tail might be used to disable the ornaments by setting the tail to be `o=0` or `ornaments=0`. The head should be as described above, say

If the variable is not set, or if the head of space-separated list is empty, the best available package is loaded.

```
export "PERL_RL=Perl o=0"      # Use Perl ReadLine without ornaments
export "PERL_RL= o=0"          # Use best available ReadLine without ornaments
```

(Note that processing of `PERL_RL` for ornaments is in the discretion of the particular used `Term::ReadLine::*` package).

no real package is found, substitutes stubs instead of basic functions."

**NAME**

```
Test − provides a simple framework for writing test scripts
```

**SYNOPSIS**

```
use strict;
use Test;
BEGIN { plan tests => 13, todo => [3,4] }

ok(0); # failure
ok(1); # success

ok(0); # ok, expected failure (see todo list, above)
ok(1); # surprise success!

ok(0,1);             # failure: '0' ne '1'
ok('broke','fixed'); # failure: 'broke' ne 'fixed'
ok('fixed','fixed'); # success: 'fixed' eq 'fixed'

ok(sub { 1+1 }, 2);  # success: '2' eq '2'
ok(sub { 1+1 }, 3);  # failure: '2' ne '3'
ok(0, int(rand(2));  # (just kidding! :-)

my @list = (0,0);
ok @list, 3, "\@list=".join(',',@list);      #extra diagnostics
ok 'segmentation fault', '/(?i)success/';    #regex match

skip($feature_is_missing, ...);    #do platform specific test
```

**DESCRIPTION**

Test::Harness expects to see particular output when it executes tests. This module aims to make writing proper test scripts just a little bit easier (and less error prone :−).

**TEST TYPES**

- NORMAL TESTS

  These tests are expected to succeed. If they don't, something's screwed up!

- SKIPPED TESTS

  Skip tests need a platform specific feature that might or might not be available. The first argument should evaluate to true if the required feature is NOT available. After the first argument, skip tests work exactly the same way as do normal tests.

- TODO TESTS

  TODO tests are designed for maintaining an executable TODO list. These tests are expected NOT to succeed (otherwise the feature they test would be on the new feature list, not the TODO list).

  Packages should NOT be released with successful TODO tests. As soon as a TODO test starts working, it should be promoted to a normal test and the newly minted feature should be documented in the release notes.

**ONFAIL**

```
BEGIN { plan test => 4, onfail => sub { warn "CALL 911!" } }
```

The test failures can trigger extra diagnostics at the end of the test run. `onfail` is passed an array ref of hash refs that describe each test failure. Each hash will contain at least the following fields: package, repetition, and result. (The file, line, and test number are not included because their correspondance to a particular test is fairly weak.) If the test had an expected value or a diagnostic string, these will also be included.

This optional feature might be used simply to print out the version of your package and/or how to report problems. It might also be used to generate extremely sophisticated diagnostics for a particular test failure. It's not a panacea, however. Core dumps or other unrecoverable errors will prevent the `onfail` hook from running. (It is run inside an END block.) Besides, `onfail` is probably over-kill in the majority of cases. (Your test code should be simpler than the code it is testing, yes?)

**SEE ALSO**

the *Test::Harness* manpage and various test coverage analysis tools.

**AUTHOR**

**NAME**

Test::Harness – run perl standard test scripts with statistics

**SYNOPSIS**

use Test::Harness;

*runtests* (@tests);

**DESCRIPTION**

(By using the the *Test* manpage module, you can write test scripts without knowing the exact output this module expects. However, if you need to know the specifics, read on!)

Perl test scripts print to standard output `"ok N"` for each single test, where `N` is an increasing sequence of integers. The first line output by a standard test script is `"1..M"` with `M` being the number of tests that should be run within the test script. *Test::Harness::runtests* (@tests) runs all the testscripts named as arguments and checks standard output for the expected `"ok N"` strings.

After all tests have been performed, *runtests()* prints some performance statistics that are computed by the Benchmark module.

### The test script output

Any output from the testscript to standard error is ignored and bypassed, thus will be seen by the user. Lines written to standard output containing `/^(not\s+)?ok\b/` are interpreted as feedback for *runtests()*. All other lines are discarded.

It is tolerated if the test numbers after `ok` are omitted. In this case Test::Harness maintains temporarily its own counter until the script supplies test numbers again. So the following test script

```
print <<END;
1..6
not ok
ok
not ok
ok
ok
END
```

will generate

```
FAILED tests 1, 3, 6
Failed 3/6 tests, 50.00% okay
```

The global variable `$Test::Harness::verbose` is exportable and can be used to let *runtests()* display the standard output of the script without altering the behavior otherwise.

The global variable `$Test::Harness::switches` is exportable and can be used to set perl command line options used for running the test *script* (s). The default value is `-w`.

If the standard output line contains substring `# Skip` (with variations in spacing and case) after `ok` or `ok NUMBER`, it is counted as a skipped test. If the whole testscript succeeds, the count of skipped tests is included in the generated output.

**EXPORT**

`&runtests` is exported by Test::Harness per default.

**DIAGNOSTICS**

`All tests successful.\nFiles=%d, Tests=%d, %s`

If all tests are successful some statistics about the performance are printed.

`FAILED tests %s\n\tFailed %d/%d tests, %.2f%% okay.`
: For any single script that has failing subtests statistics like the above are printed.

`Test returned status %d (wstat %d)`
: Scripts that return a non-zero exit status, both `$? >> 8` and `$?` are printed in a message similar to the above.

`Failed 1 test, %.2f%% okay. %s`

`Failed %d/%d tests, %.2f%% okay. %s`
: If not all tests were successful, the script dies with one of the above messages.

**ENVIRONMENT**

Setting `HARNESS_IGNORE_EXITCODE` makes it ignore the exit status of child processes.

**SEE ALSO**

the *Test* manpage for writing test scripts and also the *Benchmark* manpage for the underlying timing routines.

**AUTHORS**

Either Tim Bunce or Andreas Koenig, we don't know. What we know for sure is, that it was inspired by Larry Wall's TEST script that came with perl distributions for ages. Numerous anonymous contributors exist. Current maintainer is Andreas Koenig.

**BUGS**

Test::Harness uses $^X to determine the perl binary to run the tests with. Test scripts running via the shebang (`#!`) line may not be portable because $^X is not consistent for shebang scripts across platforms. This is no problem when Test::Harness is run with an absolute path to the perl binary or when $^X can be found in the path.

**NAME**

abbrev – create an abbreviation table from a list

**SYNOPSIS**

```
use Text::Abbrev;
abbrev $hashref, LIST
```

**DESCRIPTION**

Stores all unambiguous truncations of each element of LIST as keys key in the associative array referenced to by `$hashref`. The values are the original list elements.

**EXAMPLE**

```
$hashref = abbrev qw(list edit send abort gripe);

%hash = abbrev qw(list edit send abort gripe);

abbrev $hashref, qw(list edit send abort gripe);

abbrev(*hash, qw(list edit send abort gripe));
```

## NAME

Text::ParseWords – parse text into an array of tokens or array of arrays

## SYNOPSIS

```
use Text::ParseWords;
@lists = &nested_quotewords($delim, $keep, @lines);
@words = &quotewords($delim, $keep, @lines);
@words = &shellwords(@lines);
@words = &parse_line($delim, $keep, $line);
@words = &old_shellwords(@lines); # DEPRECATED!
```

## DESCRIPTION

The &*nested_quotewords()* and &*quotewords()* functions accept a delimiter (which can be a regular expression) and a list of lines and then breaks those lines up into a list of words ignoring delimiters that appear inside quotes. &*quotewords()* returns all of the tokens in a single long list, while &*nested_quotewords()* returns a list of token lists corresponding to the elements of `@lines`. &*parse_line()* does tokenizing on a single string. The &**quotewords()* functions simply call &*parse_lines()*, so if you're only splitting one line you can call &*parse_lines()* directly and save a function call.

The `$keep` argument is a boolean flag. If true, then the tokens are split on the specified delimiter, but all other characters (quotes, backslashes, etc.) are kept in the tokens. If `$keep` is false then the &**quotewords()* functions remove all quotes and backslashes that are not themselves backslash-escaped or inside of single quotes (i.e., &*quotewords()* tries to interpret these characters just like the Bourne shell). NB: these semantics are significantly different from the original version of this module shipped with Perl 5.000 through 5.004. As an additional feature, `$keep` may be the keyword "delimiters" which causes the functions to preserve the delimiters in each string as tokens in the token lists, in addition to preserving quote and backslash characters.

&*shellwords()* is written as a special case of &*quotewords()*, and it does token parsing with whitespace as a delimiter-- similar to most Unix shells.

## EXAMPLES

The sample program:

```
use Text::ParseWords;
@words = &quotewords('\s+', 0, q{this   is "a test" of\ quotewords \"for you});
$i = 0;
foreach (@words) {
    print "$i: <$_>\n";
    $i++;
}
```

produces:

```
0: <this>
1: <is>
2: <a test>
3: <of quotewords>
4: <"for>
5: <you>
```

demonstrating:

0 a simple word

1 multiple spaces are skipped because of our $delim

2 use of quotes to include a space in a word

3 use of a backslash to include a space in a word

4 use of a backslash to remove the special meaning of a double-quote

5 another simple word (note the lack of effect of the backslashed double-quote)

Replacing `&quotewords('\s+', 0, q{this   is...})` with `&shellwords(q{this is...})` is a simpler way to accomplish the same thing.

**AUTHORS**

Maintainer is Hal Pomeranz <pomeranz@netcom.com>, 1994-1997 (Original author unknown). Much of the code for &*parse_line()* (including the primary regexp) from Joerk Behrends <jbehrends@multimediaproduzenten.de>.

Examples section another documentation provided by John Heidemann <johnh@ISI.EDU>

Bug reports, patches, and nagging provided by lots of folks-- thanks everybody! Special thanks to Michael Schwern <schwern@envirolink.org> for assuring me that a &*nested_quotewords()* would be useful, and to Jeff Friedl <jfriedl@yahoo-inc.com> for telling me not to worry about error-checking (sort of-- you had to be there).

## NAME

Text::Soundex – Implementation of the Soundex Algorithm as Described by Knuth

## SYNOPSIS

```
use Text::Soundex;

$code = soundex $string;            # get soundex code for a string
@codes = soundex @list;             # get list of codes for list of strings

# set value to be returned for strings without soundex code

$soundex_nocode = 'Z000';
```

## DESCRIPTION

This module implements the soundex algorithm as described by Donald Knuth in Volume 3 of **The Art of Computer Programming**. The algorithm is intended to hash words (in particular surnames) into a small space using a simple model which approximates the sound of the word when spoken by an English speaker. Each word is reduced to a four character string, the first character being an upper case letter and the remaining three being digits.

If there is no soundex code representation for a string then the value of `$soundex_nocode` is returned. This is initially set to `undef`, but many people seem to prefer an *unlikely* value like `Z000` (how unlikely this is depends on the data set being dealt with.) Any value can be assigned to `$soundex_nocode`.

In scalar context `soundex` returns the soundex code of its first argument, and in array context a list is returned in which each element is the soundex code for the corresponding argument passed to `soundex` e.g.

```
@codes = soundex qw(Mike Stok);
```

leaves `@codes` containing `('M200', 'S320')`.

## EXAMPLES

Knuth's examples of various names and the soundex codes they map to are listed below:

```
Euler, Ellery -> E460
Gauss, Ghosh -> G200
Hilbert, Heilbronn -> H416
Knuth, Kant -> K530
Lloyd, Ladd -> L300
Lukasiewicz, Lissajous -> L222
```

so:

```
$code = soundex 'Knuth';              # $code contains 'K530'
@list = soundex qw(Lloyd Gauss);      # @list contains 'L300', 'G200'
```

## LIMITATIONS

As the soundex algorithm was originally used a **long** time ago in the US it considers only the English alphabet and pronunciation.

As it is mapping a large space (arbitrary length strings) onto a small space (single letter plus 3 digits) no inference can be made about the similarity of two strings which end up with the same soundex code. For example, both `Hilbert` and `Heilbronn` end up with a soundex code of `H416`.

## AUTHOR

This code was implemented by Mike Stok (`stok@cybercom.net`) from the description given by Knuth. Ian Phillips (`ian@pipex.net`) and Rich Pinder (`rpinder@hsc.usc.edu`) supplied ideas and spotted mistakes.

**NAME**

Text::Tabs -- expand and unexpand tabs per the unix *expand*(1) and *unexpand*(1)

**SYNOPSIS**

use Text::Tabs;

$tabstop = 4; `@lines_without_tabs =` *expand*(@lines_with_tabs); `@lines_with_tabs =` *unexpand*(@lines_without_tabs);

**DESCRIPTION**

Text::Tabs does about what the unix utilities *expand*(1) and *unexpand*(1) do. Given a line with tabs in it, expand will replace the tabs with the appropriate number of spaces. Given a line with or without tabs in it, unexpand will add tabs when it can save bytes by doing so. Invisible compression with plain ascii!

**BUGS**

expand doesn't handle newlines very quickly -- do not feed it an entire document in one string. Instead feed it an array of lines.

**AUTHOR**

David Muir Sharnoff <muir@idiom.com>

## NAME

Text::Wrap – line wrapping to form simple paragraphs

## SYNOPSIS

```
use Text::Wrap

print wrap($initial_tab, $subsequent_tab, @text);

use Text::Wrap qw(wrap $columns $tabstop fill);

$columns = 132;
$tabstop = 4;

print fill($initial_tab, $subsequent_tab, @text);
print fill("", "", `cat book`);
```

## DESCRIPTION

*Text::Wrap::wrap()* is a very simple paragraph formatter. It formats a single paragraph at a time by breaking lines at word boundries. Indentation is controlled for the first line ($initial_tab) and all subsquent lines ($subsequent_tab) independently. `$Text::Wrap::columns` should be set to the full width of your output device.

*Text::Wrap::fill()* is a simple multi-paragraph formatter. It formats each paragraph separately and then joins them together when it's done. It will destory any whitespace in the original text. It breaks text into paragraphs by looking for whitespace after a newline. In other respects it acts like *wrap()*.

## EXAMPLE

```
print wrap("\t","","This is a bit of text that forms
        a normal book-style paragraph");
```

## BUGS

It's not clear what the correct behavior should be when *Wrap()* is presented with a word that is longer than a line. The previous behavior was to die. Now the word is now split at line-length.

## AUTHOR

David Muir Sharnoff <muir@idiom.com> with help from Tim Pierce and others. Updated by Jacqui Caren.

**NAME**

Tie::Array – base class for tied arrays

**SYNOPSIS**

```
package NewArray;
use Tie::Array;
@ISA = ('Tie::Array');

# mandatory methods
sub TIEARRAY { ... }
sub FETCH { ... }
sub FETCHSIZE { ... }

sub STORE { ... }        # mandatory if elements writeable
sub STORESIZE { ... }    # mandatory if elements can be added/deleted

# optional methods - for efficiency
sub CLEAR { ... }
sub PUSH { ... }
sub POP { ... }
sub SHIFT { ... }
sub UNSHIFT { ... }
sub SPLICE { ... }
sub EXTEND { ... }
sub DESTROY { ... }

package NewStdArray;
use Tie::Array;

@ISA = ('Tie::StdArray');

# all methods provided by default

package main;

$object = tie @somearray,Tie::NewArray;
$object = tie @somearray,Tie::StdArray;
$object = tie @somearray,Tie::NewStdArray;
```

**DESCRIPTION**

This module provides methods for array-tying classes. See the *perltie* manpage for a list of the functions required in order to tie an array to a package. The basic **Tie::Array** package provides stub `DELETE` and `EXTEND` methods, and implementations of `PUSH`, `POP`, `SHIFT`, `UNSHIFT`, `SPLICE` and `CLEAR` in terms of basic `FETCH`, `STORE`, `FETCHSIZE`, `STORESIZE`.

The **Tie::StdArray** package provides efficient methods required for tied arrays which are implemented as blessed references to an "inner" perl array. It inherits from **Tie::Array**, and should cause tied arrays to behave exactly like standard arrays, allowing for selective overloading of methods.

For developers wishing to write their own tied arrays, the required methods are briefly defined below. See the the *perltie* manpage section for more detailed descriptive, as well as example code:

TIEARRAY classname, LIST

The class method is invoked by the command `tie @array, classname`. Associates an array instance with the specified class. `LIST` would represent additional arguments (along the lines of the *AnyDBM_File* manpage and compatriots) needed to complete the association. The method should return an object of a class which provides the methods below.

STORE this, index, value
: Store datum *value* into *index* for the tied array assoicated with object *this*. If this makes the array larger then class's mapping of `undef` should be returned for new positions.

FETCH this, index
: Retrieve the datum in *index* for the tied array assoicated with object *this*.

FETCHSIZE this
: Returns the total number of items in the tied array assoicated with object *this*. (Equivalent to `scalar(@array)`).

STORESIZE this, count
: Sets the total number of items in the tied array assoicated with object *this* to be *count*. If this makes the array larger then class's mapping of `undef` should be returned for new positions. If the array becomes smaller then entries beyond count should be deleted.

EXTEND this, count
: Informative call that array is likely to grow to have *count* entries. Can be used to optimize allocation. This method need do nothing.

CLEAR this
: Clear (remove, delete, ...) all values from the tied array assoicated with object *this*.

DESTROY this
: Normal object destructor method.

PUSH this, LIST
: Append elements of LIST to the array.

POP this
: Remove last element of the array and return it.

SHIFT this
: Remove the first element of the array (shifting other elements down) and return it.

UNSHIFT this, LIST
: Insert LIST elements at the begining of the array, moving existing elements up to make room.

SPLICE this, offset, length, LIST
: Perform the equivalent of `splice` on the array.

*offset* is optional and defaults to zero, negative values count back from the end of the array.

*length* is optional and defaults to rest of the array.

*LIST* may be empty.

Returns a list of the original *length* elements at *offset*.

**CAVEATS**

There is no support at present for tied `@ISA`. There is a potential conflict between magic entries needed to notice setting of `@ISA`, and those needed to implement 'tie'.

Very little consideration has been given to the behaviour of tied arrays when `$[` is not default value of zero.

**AUTHOR**

Nick Ing-Simmons <nik@tiuk.ti.com>

## NAME

Tie::Handle – base class definitions for tied handles

## SYNOPSIS

```
package NewHandle;
require Tie::Handle;

@ISA = (Tie::Handle);

sub READ { ... }            # Provide a needed method
sub TIEHANDLE { ... }       # Overrides inherited method


package main;

tie *FH, 'NewHandle';
```

## DESCRIPTION

This module provides some skeletal methods for handle-tying classes. See the *perltie* manpage for a list of the functions required in tying a handle to a package. The basic **Tie::Handle** package provides a `new` method, as well as methods `TIESCALAR`, `FETCH` and `STORE`. The `new` method is provided as a means of grandfathering, for classes that forget to provide their own `TIESCALAR` method.

For developers wishing to write their own tied-handle classes, the methods are summarized below. The the *perltie* manpage section not only documents these, but has sample code as well:

TIEHANDLE classname, LIST
: The method invoked by the command `tie *glob, classname`. Associates a new glob instance with the specified class. `LIST` would represent additional arguments (along the lines of the *AnyDBM_File* manpage and compatriots) needed to complete the association.

WRITE this, scalar, length, offset
: Write *length* bytes of data from *scalar* starting at *offset*.

PRINT this, LIST
: Print the values in *LIST*

PRINTF this, format, LIST
: Print the values in *LIST* using *format*

READ this, scalar, length, offset
: Read *length* bytes of data into *scalar* starting at *offset*.

READLINE this
: Read a single line

GETC this
: Get a single character

DESTROY this
: Free the storage associated with the tied handle referenced by *this*. This is rarely needed, as Perl manages its memory quite well. But the option exists, should a class wish to perform specific actions upon the destruction of an instance.

## MORE INFORMATION

The the *perltie* manpage section contains an example of tying handles.

## NAME

Tie::Hash, Tie::StdHash − base class definitions for tied hashes

## SYNOPSIS

```
package NewHash;
require Tie::Hash;

@ISA = (Tie::Hash);

sub DELETE { ... }          # Provides needed method
sub CLEAR { ... }           # Overrides inherited method


package NewStdHash;
require Tie::Hash;

@ISA = (Tie::StdHash);

# All methods provided by default, define only those needing overrides
sub DELETE { ... }


package main;

tie %new_hash, 'NewHash';
tie %new_std_hash, 'NewStdHash';
```

## DESCRIPTION

This module provides some skeletal methods for hash-tying classes. See the *perltie* manpage for a list of the functions required in order to tie a hash to a package. The basic **Tie::Hash** package provides a `new` method, as well as methods `TIEHASH`, `EXISTS` and `CLEAR`. The **Tie::StdHash** package provides most methods required for hashes in the *perltie* manpage. It inherits from **Tie::Hash**, and causes tied hashes to behave exactly like standard hashes, allowing for selective overloading of methods. The `new` method is provided as grandfathering in the case a class forgets to include a `TIEHASH` method.

For developers wishing to write their own tied hashes, the required methods are briefly defined below. See the the *perltie* manpage section for more detailed descriptive, as well as example code:

TIEHASH classname, LIST
: The method invoked by the command `tie %hash, classname`. Associates a new hash instance with the specified class. `LIST` would represent additional arguments (along the lines of the *AnyDBM_File* manpage and compatriots) needed to complete the association.

STORE this, key, value
: Store datum *value* into *key* for the tied hash *this*.

FETCH this, key
: Retrieve the datum in *key* for the tied hash *this*.

FIRSTKEY this
: Return the (key, value) pair for the first key in the hash.

NEXTKEY this, lastkey
: Return the next key for the hash.

EXISTS this, key
: Verify that *key* exists with the tied hash *this*.

DELETE this, key
: Delete the key *key* from the tied hash *this*.

CLEAR this
: Clear all values from the tied hash *this*.

**CAVEATS**

The the *perltie* manpage documentation includes a method called `DESTROY` as a necessary method for tied hashes. Neither **Tie::Hash** nor **Tie::StdHash** define a default for this method. This is a standard for class packages, but may be omitted in favor of a simple default.

**MORE INFORMATION**

The packages relating to various DBM−related implemetations (*DB_File*, *NDBM_File*, etc.) show examples of general tied hashes, as does the the *Config* manpage module. While these do not utilize **Tie::Hash**, they serve as good working examples.

## NAME

Tie::RefHash – use references as hash keys

## SYNOPSIS

```
require 5.004;
use Tie::RefHash;
tie HASHVARIABLE, 'Tie::RefHash', LIST;

untie HASHVARIABLE;
```

## DESCRIPTION

This module provides the ability to use references as hash keys if you first `tie` the hash variable to this module.

It is implemented using the standard perl TIEHASH interface. Please see the `tie` entry in *perlfunc* (1) and *perltie* (1) for more information.

## EXAMPLE

```
use Tie::RefHash;
tie %h, 'Tie::RefHash';
$a = [];
$b = {};
$c = \*main;
$d = \"gunk";
$e = sub { 'foo' };
%h = ($a => 1, $b => 2, $c => 3, $d => 4, $e => 5);
$a->[0] = 'foo';
$b->{foo} = 'bar';
for (keys %h) {
   print ref($_), "\n";
}
```

## AUTHOR

Gurusamy Sarathy gsar@umich.edu

## VERSION

Version 1.2 15 Dec 1996

## SEE ALSO

*perl* (1), *perlfunc* (1), *perltie* (1)

## NAME

Tie::Scalar, Tie::StdScalar – base class definitions for tied scalars

## SYNOPSIS

```
package NewScalar;
require Tie::Scalar;

@ISA = (Tie::Scalar);

sub FETCH { ... }           # Provide a needed method
sub TIESCALAR { ... }       # Overrides inherited method


package NewStdScalar;
require Tie::Scalar;

@ISA = (Tie::StdScalar);

# All methods provided by default, so define only what needs be overridden
sub FETCH { ... }


package main;

tie $new_scalar, 'NewScalar';
tie $new_std_scalar, 'NewStdScalar';
```

## DESCRIPTION

This module provides some skeletal methods for scalar-tying classes. See the *perltie* manpage for a list of the functions required in tying a scalar to a package. The basic **Tie::Scalar** package provides a `new` method, as well as methods `TIESCALAR`, `FETCH` and `STORE`. The **Tie::StdScalar** package provides all the methods specified in the *perltie* manpage. It inherits from **Tie::Scalar** and causes scalars tied to it to behave exactly like the built-in scalars, allowing for selective overloading of methods. The `new` method is provided as a means of grandfathering, for classes that forget to provide their own `TIESCALAR` method.

For developers wishing to write their own tied-scalar classes, the methods are summarized below. The the *perltie* manpage section not only documents these, but has sample code as well:

TIESCALAR classname, LIST

The method invoked by the command `tie $scalar, classname`. Associates a new scalar instance with the specified class. `LIST` would represent additional arguments (along the lines of the *AnyDBM_File* manpage and compatriots) needed to complete the association.

FETCH this

Retrieve the value of the tied scalar referenced by *this*.

STORE this, value

Store data *value* in the tied scalar referenced by *this*.

DESTROY this

Free the storage associated with the tied scalar referenced by *this*. This is rarely needed, as Perl manages its memory quite well. But the option exists, should a class wish to perform specific actions upon the destruction of an instance.

## MORE INFORMATION

The the *perltie* manpage section uses a good example of tying scalars by associating process IDs with priority.

## NAME

Tie::SubstrHash − Fixed-table-size, fixed-key-length hashing

## SYNOPSIS

```
require Tie::SubstrHash;

tie %myhash, 'Tie::SubstrHash', $key_len, $value_len, $table_size;
```

## DESCRIPTION

The **Tie::SubstrHash** package provides a hash-table-like interface to an array of determinate size, with constant key size and record size.

Upon tying a new hash to this package, the developer must specify the size of the keys that will be used, the size of the value fields that the keys will index, and the size of the overall table (in terms of key-value pairs, not size in hard memory). *These values will not change for the duration of the tied hash.* The newly-allocated hash table may now have data stored and retrieved. Efforts to store more than `$table_size` elements will result in a fatal error, as will efforts to store a value not exactly `$value_len` characters in length, or reference through a key not exactly `$key_len` characters in length. While these constraints may seem excessive, the result is a hash table using much less internal memory than an equivalent freely-allocated hash table.

## CAVEATS

Because the current implementation uses the table and key sizes for the hashing algorithm, there is no means by which to dynamically change the value of any of the initialization parameters.

**NAME**

Time::gmtime − by-name interface to Perl's built-in *gmtime()* function

**SYNOPSIS**

```
use Time::gmtime;
$gm = gmtime();
printf "The day in Greenwich is %s\n",
   (qw(Sun Mon Tue Wed Thu Fri Sat Sun))[ gm->wday() ];

use Time::gmtime w(:FIELDS;
printf "The day in Greenwich is %s\n",
   (qw(Sun Mon Tue Wed Thu Fri Sat Sun))[ gm_wday() ];

$now = gmctime();

use Time::gmtime;
use File::stat;
$date_string = gmctime(stat($file)->mtime);
```

**DESCRIPTION**

This module's default exports override the core *gmtime()* function, replacing it with a version that returns "Time::tm" objects. This object has methods that return the similarly named structure field name from the C's tm structure from *time.h*; namely sec, min, hour, mday, mon, year, wday, yday, and isdst.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `tm_` in front their method names. Thus, `$tm_obj->mday()` corresponds to `$tm_mday` if you import the fields.

The *gmctime()* funtion provides a way of getting at the scalar sense of the original *CORE::gmtime()* function.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

**NOTE**

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

**AUTHOR**

Tom Christiansen

## NAME

Time::Local − efficiently compute time from local and GMT time

## SYNOPSIS

```
$time = timelocal($sec,$min,$hours,$mday,$mon,$year);
$time = timegm($sec,$min,$hours,$mday,$mon,$year);
```

## DESCRIPTION

These routines are quite efficient and yet are always guaranteed to agree with *localtime()* and *gmtime()*. We manage this by caching the start times of any months we've seen before. If we know the start time of the month, we can always calculate any time within the month. The start times themselves are guessed by successive approximation starting at the current time, since most dates seen in practice are close to the current date. Unlike algorithms that do a binary search (calling gmtime once for each bit of the time value, resulting in 32 calls), this algorithm calls it at most 6 times, and usually only once or twice. If you hit the month cache, of course, it doesn't call it at all.

timelocal is implemented using the same cache. We just assume that we're translating a GMT time, and then fudge it when we're done for the timezone and daylight savings arguments. The timezone is determined by examining the result of *localtime*(0) when the package is initialized. The daylight savings offset is currently assumed to be one hour.

Both routines return −1 if the integer limit is hit. I.e. for dates after the 1st of January, 2038 on most machines.

## NAME

Time::localtime – by-name interface to Perl's built-in *localtime()* function

## SYNOPSIS

```
use Time::localtime;
printf "Year is %d\n", localtime->year() + 1900;

$now = ctime();

use Time::localtime;
use File::stat;
$date_string = ctime(stat($file)->mtime);
```

## DESCRIPTION

This module's default exports override the core *localtime()* function, replacing it with a version that returns "Time::tm" objects. This object has methods that return the similarly named structure field name from the C's tm structure from *time.h*; namely sec, min, hour, mday, mon, year, wday, yday, and isdst.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `tm_` in front their method names. Thus, `$tm_obj->mday()` corresponds to `$tm_mday` if you import the fields.

The *ctime()* funtion provides a way of getting at the scalar sense of the original *CORE::localtime()* function.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

## NOTE

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

## AUTHOR

Tom Christiansen

## NAME

Time::tm – internal object used by Time::gmtime and Time::localtime

## SYNOPSIS

Don't use this module directly.

## DESCRIPTION

This module is used internally as a base class by Time::localtime And Time::gmtime functions. It creates a Time::tm struct object which is addressable just like's C's tm structure from *time.h*; namely with sec, min, hour, mday, mon, year, wday, yday, and isdst.

This class is an internal interface only.

## AUTHOR

Tom Christiansen

## NAME

UNIVERSAL – base class for ALL classes (blessed references)

## SYNOPSIS

```
$io = $fd->isa("IO::Handle");
$sub = $obj->can('print');

$yes = UNIVERSAL::isa($ref, "HASH");
```

## DESCRIPTION

`UNIVERSAL` is the base class which all bless references will inherit from, see the *perlobj* manpage

`UNIVERSAL` provides the following methods

isa ( TYPE )

`isa` returns *true* if `REF` is blessed into package `TYPE` or inherits from package `TYPE`.

`isa` can be called as either a static or object method call.

can ( METHOD )

`can` checks if the object has a method called `METHOD`. If it does then a reference to the sub is returned. If it does not then *undef* is returned.

`can` can be called as either a static or object method call.

VERSION ( [ REQUIRE ] )

`VERSION` will return the value of the variable `$VERSION` in the package the object is blessed into. If `REQUIRE` is given then it will do a comparison and die if the package version is not greater than or equal to `REQUIRE`.

`VERSION` can be called as either a static or object method call.

The `isa` and `can` methods can also be called as subroutines

UNIVERSAL::isa ( VAL, TYPE )

`isa` returns *true* if the first argument is a reference and either of the following statements is true.

`VAL` is a blessed reference and is blessed into package `TYPE` or inherits from package `TYPE`

`VAL` is a reference to a `TYPE` of perl variable (er 'HASH')

UNIVERSAL::can ( VAL, METHOD )

If `VAL` is a blessed reference which has a method called `METHOD`, `can` returns a reference to the subroutine. If `VAL` is not a blessed reference, or if it does not have a method `METHOD`, *undef* is returned.

These subroutines should *not* be imported via `use UNIVERSAL qw(...)`. If you want simple local access to them you can do

```
*isa = \&UNIVERSAL::isa;
```

to import isa into your package.

**NAME**

User::grent − by-name interface to Perl's built-in getgr*() functions

**SYNOPSIS**

```
use User::grent;
$gr = getgrgid(0) or die "No group zero";
if ( $gr->name eq 'wheel' && @{$gr->members} > 1 ) {
    print "gid zero name wheel, with other members";
}

use User::grent qw(:FIELDS;
getgrgid(0) or die "No group zero";
if ( $gr_name eq 'wheel' && @gr_members > 1 ) {
    print "gid zero name wheel, with other members";
}

$gr = getgr($whoever);
```

**DESCRIPTION**

This module's default exports override the core *getgrent()*, *getgruid()*, and *getgrnam()* functions, replacing them with versions that return "User::grent" objects. This object has methods that return the similarly named structure field name from the C's passwd structure from *grp.h*; namely name, passwd, gid, and members (not mem). The first three return scalars, the last an array reference.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `gr_`. Thus, `$group_obj->gid()` corresponds to `$gr_gid` if you import the fields. Array references are available as regular array variables, so `@{ $group_obj->members() }` would be simply `@gr_members`.

The *getpw()* funtion is a simple front-end that forwards a numeric argument to *getpwuid()* and the rest to *getpwnam()*.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

**NOTE**

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

**AUTHOR**

Tom Christiansen

**NAME**

User::pwent − by-name interface to Perl's built-in getpw*() functions

**SYNOPSIS**

```
use User::pwent;
$pw = getpwnam('daemon') or die "No daemon user";
if ( $pw->uid == 1 && $pw->dir =~ m#^/(bin|tmp)?$# ) {
    print "gid 1 on root dir";
}

use User::pwent qw(:FIELDS);
getpwnam('daemon') or die "No daemon user";
if ( $pw_uid == 1 && $pw_dir =~ m#^/(bin|tmp)?$# ) {
    print "gid 1 on root dir";
}

$pw = getpw($whoever);
```

**DESCRIPTION**

This module's default exports override the core *getpwent()*, *getpwuid()*, and *getpwnam()* functions, replacing them with versions that return "User::pwent" objects. This object has methods that return the similarly named structure field name from the C's passwd structure from *pwd.h*; namely name, passwd, uid, gid, quota, comment, gecos, dir, and shell.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `pw_` in front their method names. Thus, `$passwd_obj->shell()` corresponds to `$pw_shell` if you import the fields.

The *getpw()* funtion is a simple front-end that forwards a numeric argument to *getpwuid()* and the rest to *getpwnam()*.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

**NOTE**

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

**AUTHOR**

Tom Christiansen