PERL

a language by Larry Wall

Practical Extraction and Report Language

or

Pathologically Eclectic Rubbish Lister

Tom Christiansen

Overview

- What is *Perl*: features, where to get it, preview
- Data Types: scalars and arrays
- Operators
- Flow Control
- Regular Expressions
- I/O: regular I/O, system functions, directory access, formatted I/O
- Functions and Subroutines: built-in array and string functions
- Esoterica: debugging, packages, command line options
- Examples

What is Perl?

- An interpreted language that looks a lot like C with built-in *sed*, *awk*, and *sh*, as well as bits of *csh*, Pascal, FORTRAN, BASIC-PLUS thrown in.
- Highly optimized for manipulating printable text, but also able to handle binary data.
- Especially suitable for system management tasks due to interfaces to most common system calls.
- Rich enough for most general programming tasks.
- *"A shell for C programmers."* [Larry Wall]

Where to find out more?

- *Learning perl* by Randal Schwartz (the llama book)
- *Programming perl* by Larry Wall and Randal Schwartz (the camel book)
- The *Quick Reference Guide* by Johan Vromans
- The man pages
- comp.lang.perl

Features

- Easy to learn because much derives from existing tools.
- More rapid program development because it's an interpreter
- Faster execution than shell script equivalents.
- More powerful than *sed*, *awk*, or *sh*; *a2p* and *s2p* translators supplied for your old scripts.
- Portable across many different architectures.
- Absence of arbitrary limits like string length.
- Fits nicely into UNIX tool and filter philosophy.
- It's free!

Preview

- It's not for nothing that *perl* is sometimes called the "pathologically eclectic rubbish lister." Before you drown in a deluge of features, here's a simple example to whet your appetites that demonstrates the principal features of the language, all of which have been present since version 1.

```
while (<>) {
    next if /^#/;
    ($x, $y, $z) = /(\S+)\s+(\d\d\d)\s+(foo|bar)/;
    $x =~ tr/a-z/A-Z/;
    $seen{$x}++;
    $z =~ s/foo/fear/ && $scared++;
    printf "%s %08x %-10s\n", $z, $y, $x
        if $seen{$x} > $y;
}
```

Data Types

- Basic data types are scalars, indexed arrays of scalars, and associative arrays of scalars.
- Scalars themselves are either string, numeric, or boolean, depending on context. Values of 0 (zero) and "" (null string) are false; all else is true.
- Type of variable determined by leading special character.

$	scalar
@	indexed array (lists)
%	associative array
&	function

- All data types have their own separate namespaces, as do labels, functions, and file and directory handles.

Data Types (scalars)

- Use a $ to indicate a scalar value

```
$foo = 3.14159;
$foo = 'red';
$foo = "was $foo before";
$host = `hostname`;           # note backticks
($foo, $bar, $glarch) = ('red', 'blue', 'green');
($foo, $bar) = ($bar, $foo);  # exchange
```

Special Scalar Variables

- Special scalars are named with punctuation (except **$0**). Examples are

$0	name of the currently executing script
$_	default for pattern operators and implicit I/O
$$	the current pid
$!	the current system error message from *errno*
$?	status of last ‘backtick‘, pipe, or system
$|	whether output is buffered
$.	the current line number of last input
$[	array base, 0 by default; *awk* uses 1
$<	the real uid of the process
$>	the effective uid of the process

Data types (arrays)

- Indexed arrays (lists); $ for one scalar element, @ for all

```
$foo[$i+2] = 3;         # set one element to 3
@foo = ( 1, 3, 5 );     # init whole array
@foo = ( ) ;            # initialize empty array
@foo = @bar;            # copy whole @array
@foo = @bar[$i..$i+5];  # copy slice of @array
```

- $#ARRAY is index of highest subscript, so the script's name is **$0** and its arguments run from $ARGV[0] through $ARGV[$#ARGV], inclusive.

Data types (arrays)

- Associative (hashed) arrays; $ for one scalar element, % for all

```
# 23 more green frogs
    $frogs{'green'} += 23;
# multi-dim array
    $location{$x, $y, $z} = 'troll';
# copy whole %array
    %foo = %bar;
    @frogs{'green', 'blue', 'yellow'} = (3, 6, 9);
```


Special Array Variables

@ARGV	command line arguments
@INC	search path for files called with **do**
@_	default for **split** and subroutine parameters
%ENV	the current enviroment; e.g. $ENV{'HOME'}
%SIG	used to set signal handlers

```
sub trapped {
    print STDERR "Interrupted\007\n";
    exit 1;
}
$SIG{'INT'} = 'trapped';
```

Operators

Perl uses all of C's operators except for type casting and **'&'** and **'*'** as address operators, plus these

- exponentiation: **, **=
- range operator: **..**

```
$inheader = 1 if /^From / .. /^$/;
if (1..10) { do foo(); }
for $i (60..75) { do foo($i); }
@new = @old[30..50];
```

- string concatenation: **.**, **.=**

```
$x = $y . &frob(@list) . $z;
$x .= "\n";
```


Operators (continued)

- string repetition: **x, x=**

```
$bar = '-' x 72; # row of 72 dashes
```

- string tests: **eq**, **ne**, **lt**, **gt**, **le**, **ge**

```
if ($x eq 'foo') { }
if ($x ge 'red' ) { }
```

- file test operators like augmented */bin/test* tests work on strings or filehandles

```
if (-e $file) { }     # file exists
if (-z $file) { }     # zero length
if (-O LOG) { }       # LOG owned by real uid
die "$file not a text file" unless -T $file;
```

Flow Control

- Unlike C, blocks always require enclosing braces {}
- **unless** and **until** are just **if** and **while** negated

```
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK else BLOCK
while (EXPR) BLOCK
do BLOCK while EXPR
for (EXPR; EXPR; EXPR) BLOCK
foreach $VAR (LIST) BLOCK
```


Flow Control (continued)

- For readability, **if**, **unless**, **while**, and **until** may be used as trailing statement modifiers as in BASIC-PLUS

```
return -1 unless $x > 0;
```

- Use **next** and **last** rather than C's **continue** and **break**
- **redo** restarts the current iteration, ignoring the loop test
- Blocks (and **next**, **last**, and **redo**) take optional labels for clearer loop control, avoiding the use of **goto** to exit nested loops.
- No **switch** statement, but it's easy to roll your own

Flow Control (continued)

- **do** takes 3 forms
  - execute a block

```
do { $x += $a[$i++] } until $i > $j;
```

  - execute a subroutine

```
do foo($x, $y);
```

  - execute a file in current context

```
do 'subroutines.pl';
```


Regular Expressions

- Understands *egrep* regexps, plus

\w, \W	alphanumerics plus _ (and negation)
\d, \D	digits (and negation)
\s, \S	white space (and negation)
\b, \B	word boundaries (and negation)

- C-style escapes recognized, like **\t, \n, \034**

Regular Expressions (continued)

- Don't escape these characters for their special meaning: **( )** | { } +
- Character classes may contain metas, e.g. [**w.$**]
- Special variables: **$&** means all text matched, **$`** is text before match, **$'** is text after match.
- Use **1** .. **9** within rexprs; **$1** .. **$9** outside

```
if (/^this (red|blue|green) (bat|ball) is \1/)
    { ($color, $object) = ($1, $2); }
($color, $object) =
    /^this (red|blue|green) (bat|ball) is \1/;
```


Regular Expressions (continued)

- Substitute and translation operators are like *sed*'s **s** and **y**.

```
s/alpha/beta/;
s/(.)\1/$1/g;
y/A-Z/a-z/;
```

- Use =~ and !~ to match against variables

```
if ($foo !~ /^\w+$/) { exit 1; }
$foo =~ s/\btexas\b/TX/i;
```

I/O

- Filehandles have their own distinct namespaces, but are typically all upper case for clarity. Pre-defined filehandles are STDIN, STDOUT, STDERR.
- Mentioning a filehandle in angle brackets reads next line in scalar context, all lines in an array context; newlines are left intact.

```
$line = <TEMP>;
@lines = <TEMP>;
```

- <> means all files supplied on command line (or STDIN if none). When used this way, $ARGV is the current filename.

I/O (continued)

- When used in a **while** construct, input lines are automatically assigned to the **$_** variable.
- Usually iterate over file a line at a time, assigning to **$_** each time and using that as the default operand.

```
while ( <> ) {
    next if /^#/;       # skip comments
    s/left/right/g;     # global substitute
    print;              # print $_
}
```

I/O (continued)

- If not using the pseudo-file <>, open a filehandle:

```
open (PWD,      "/etc/passwd");
open (TMP,      ">/tmp/foobar.$$");
open (LOG,      ">>logfile");
open (TOPIPE,   "| lpr");
open (FROMPIPE, "/usr/etc/netstat -a |");
```


I/O (continued)

- May also use **getc** for character I/O and **read** for raw I/O
- Access to **eof**, **seek**, **close**, **flock**, **ioctl**, **fcntl**, and **select** calls for use with filehandles.
- Access to **mkdir**, **rmdir**, **chmod**, **chown**, **link**, **symlink** (if supported), **stat**, **rename**, **unlink** calls for use with filenames.
- Pass **printf** a filehandle as its first argument unless printing to STDOUT

```
printf LOG "%-8s %s: weird bits: %08x\n",
    $program, &ctime, $bits;
```

- Associative arrays may be bound to **dbm** files with **tie()**

System Functions

A plethora of functions from the C library are provided as built-ins, including most system calls. These include

- **chdir**, **chroot**, **exec**, **exit**, **fork**, **getlogin**, **getpgrp**, **getppid**, **kill**, **setpgrp**, **setpriority**, **sleep**, **syscall**, **system**, **times**, **umask**, **wait**.
- If your system has Berkeley-style networking, **bind**, **connect**, **send**, **getsockname**, **getsockopt**, **getpeername**, **recv**, **listen**, **socket**, **socketpair**.
- **getpw***, **getgr***, **gethost***, **getnet***, **getserv***, and **getproto***.
- **pack** and **unpack** can be used for manipulating binary data.

Directory Access

Three methods of accessing directories are provided.

- You may open a pipe from */bin/ls* like this:

```
open(FILES,"/bin/ls *.c |");
while ($file = <FILES>) { chop($file); ... }
```

- The directory-reading routines are provided as built-ins and operate on directory handles. Supported routines are **opendir**, **readdir**, **closedir**, **seekdir**, **telldir**, and **rewinddir**.

Directory Access (continued)

- The easiest way is to use *perl*'s file globbing notation. A string enclosed in angle brackets containing shell metacharacters evaluates to a list of matching filenames.

```
foreach $x ( <*.[ch]> ) { rename($x, "$x.old"); }
chmod 0644, <*.c>;
```


Subroutines

- Subroutines called either with **'do'** operator or with **'&'**. Any of the three principal data types may be passed as parameters or used as a return value.

```
do foo(1.43);
do foo(@list)
$x = &foo('red', 3, @others);
@list = &foo(@olist);
%foo = &foo($foo, @foo);
```

Subroutines (continued)

- Parameters are received by the subroutine in the special array @_. If desired, these can be copied to local variables. This is especially useful for recursive subroutines.

```
$result = &simple($alpha, $beta, @tutti);
sub simple {
    local($x, $y, @rest) = @_;
    local($sum, %seen);
    return $sum;
}
```

- Subroutines may also be called indirectly

```
$foo = 'some_routine';
do $foo(@list)
($x, $y, $z) = do $foo(%maps);
```

29

Formatted I/O

- Besides **printf**, formatted I/O can be done with **format** and **write** statements.
- Automatic pagination and printing of headers.
- Picture description facilitates lining up multi-line output
- Fields in picture may be left or right-justified or centered
- Multi-line text-block filling is provided, something like having a **%s** format string with a built-in pipe to **fmt**
- These special scalar variables are useful:

$%	for current page number,
$=	for current page length (default 60)
$-	for lines left on page

Formatted I/O (example)

```
# a report from a bug report form; from perl man page
format 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
.
```

Built-in Array Functions

- Indexed arrays function as lists; you can add items to or remove them from either end using these functions:

pop	remove last value from end of array
push	add values to end of array
shift	remove first value from front of array
unshift	add values to front of array

For example

```
push(@list, $bar);
push(@list, @rest);
$tos = pop(@list);
while ( $arg = shift(@ARGV) )  { }
unshift( @ARGV, 'zeroth arg', 'first arg');
```


built-in Array Functions (split and join)

- **split** breaks up a string into an array of new strings. You can **split** on arbitrary regular expressions, limit the number of fields you **split** into, and save the delimiters if you want.

```
@list = split(/[, \t]+/, $expr);
while (<PASSWD>) {
    ($login, $passwd, $uid, $gid, $gcos,
        $home, $shell) = split(/:/);
}
```

- The inverse of **split** is **join**.

```
$line = join(':', $login, $passwd, $uid,
                  $gid, $gcos, $home, $shell);
```

Built-in Array Functions (sort, grep, reverse)

- **reverse** inverts a list.

```
foreach $tick (reverse 0..10) { }
```

- **sort** returns a new array with the elements ordered according to their ASCII values. Use your own routine for different collating.

```
print sort @list;
sub numerically { $a - $b; }
print sort numerically @list;
```

- **grep** returns a new list consisting of all the elements for which a given expression is true. For example, this will delete all lines with leading pound signs:

```
@lines = grep(!/^#/, @lines);
```


Built-in Array Functions (%arrays)

For manipulating associative arrays, the **keys** and **values** functions return indexed arrays of the indices and data values respectively. **each** is used to iterate through an associative array to retrieve one **($key,$value)** pair at a time.

```
while (($key,$value) = each %array) {
    printf "%s is %s\n", $key, $value;
}

foreach $key (keys %array) {
    printf "%s is %s\n", $key, $array{$key};
}

print reverse sort values %array;
```

String functions

- Besides the powerful regular expression features, several well-known C string manipulation functions are provided, including **crypt**, **index**, **rindex**, **length**, **substr**, and **sprintf**.
- The **chop** function efficiently removes the last character from a string. It's usually used to delete the trailing newline on input lines. Like many *perl* operators, it works on **$_** if no operand is given.

```
chop($line);
chop ($host = `hostname`);
while (<STDIN>) {
    chop; ...
}
```


String functions (continued)

- The **eval** operator lets you execute dynamically generated code. For example, to process any command line arguments of the form **variable=value**, place this at the top of your script:

```
eval '$'.$1."'$2';"
    while $ARGV[0] =~ /^([A-Za-z_]+=)(.*)/ && shift;
```

The **eval** operator is also useful for run-time testing of system-dependent features which would otherwise trigger fatal errors. For example, not all systems support the **symlink** or **dbmopen**; you could test for their existence by executing the statements within an eval and testing the special variable **$@**, which contains the text of the run-time error message if anything went wrong.

Suid Scripts

- *Perl* programs can be made to run setuid, and can actually be more secure than the corresponding C program.
- Because interpreters have no guarantee that the filename they get as the first argument is the same file that was **exec**'ed, *perl* won't let you run a setuid script on a system where setuid scripts are not disabled.
- Using a dataflow tracing mechanism triggered by setuid execution, *perl* can tell what data is safe to use and what data comes from an external source and thus is "tainted."
- Tainted data may not be used directly or indirectly in any command that modifies files, directories or processes or else a fatal run-time error will result.

Packages (Modules)

- Each has own namespace.
- Access variable in another package with

```
$Package::Variable
```

- Default package is $main.
- Use packages in other files with

```
require "Module.pm";
use Module;
```

- Can be used to build abstract data types with constructors, destructors, and methods.

References

- Simplest use like C & (address of) operator:

```
$scalarref = \$foo;
$arrayref = \@ARGV;
$hashref = \%ENV;
$coderef = \&handler;
$refref = \$scalarref;
```

- Perl keeps track of reference counts, object is freed if reference count goes to zero.
- References are typed—use ref($ref) to find out type pointed to.

References to anonymous objects

Reference to anonymous array:

```
$arrayref = [1, 2, ['a', 'b', 'c']];
$arrayref->[2][1] == 'b'
```

Reference to anonymous hash:

```
$hashref = {
    'Adam'=>'Eve',
    'Clyde'=>'Bonnie',
};
```

Reference to anonymous subroutine:

```
$coderef = sub { print "Boink!\n" };
```

Dereferencing

Access value pointed to:

```
$bar = $$scalarref;
push(@$arrayref, $filename);
$$arrayref[0] = "January";
$$hashref{"KEY"} = "VALUE";
&$coderef(1,2,3);
print $$$refref;
```

or

```
$arrayref->[0] = "January";
$hashref->{"KEY"} = "VALUE";
```


Objects

- An object is just a reference which has been "approved" by some package. Its methods are just subs in that package.
- Constructors and destructors are BEGIN and END methods.
- First argument to a method is the object reference.
- Inheritance of methods is done through the @ISA list:

```
package BAR;
@ISA = (FOO);
```

Example Objects

```
package Foo;
sub new {
    my $type = shift;
    my %params = @_;
    my $self = {};
    $self->{'High'} = $params{'High'};
    $self->{'Low'}  = $params{'Low'};
    bless $self;
}
package main;
$a = new Foo ( 'High' => 42, 'Low' => 11 );
print "High=$a->{'High'}\n";
print "Low=$a->{'Low'}\n";
```


Command Line Options

The following are the more important command line switches recognized by *perl*:

−v	print out version string
−w	issue warnings about error-prone constructs
−d	run script under the debugger
−e	like *sed*: used to enter single command lines
−n	loop around input like *sed* **-n**
−p	as with **-n** but print out each line
−i	edit files in place
−a	turn on autosplit mode (like *awk*) into **@F** array
−P	call C pre-processor on script

Examples: Command Line

```
# output current version
   perl -v
# simplest perl program
   perl -e 'print "hello, world.\n";'
# useful at end of "find foo -print"
   perl -n -e 'chop;unlink;'
# add first and last columns (filter)
   perl -a -n -e 'print $F[0] + $F[$#F]. "\n";'
# in-place edit of *.c files changing all foo to bar
   perl -p -i -e 's/\bfoo\b/bar/g;' *.c
# run a script under the debugger
   perl -d myscript
```



```
#!/usr/bin/perl
#  Print out today's yearday on stdout
($sec, $min, $hour, $day, $mon, $year, @rest) = gmtime();
if ($year == 1996) {        # a leap year
    @months = (31, 29, 31, 30, 31, 30,
               31, 31, 30, 31, 30, 31);
} else {
    @months = (31, 28, 31, 30, 31, 30,
               31, 31, 30, 31, 30, 31);
}
$yearday = 0;
for ($m=0; $m < $mon; $m++) {
    $yearday += $months[$m];
}
$yearday += $day;
print $yearday . "\n";
```