NAME

perlfaq − frequently asked questions about Perl ($Date: 1997/03/17 22:17:56 $)

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 training, 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. Mail sent to the old perlfaq alias will merely cause the FAQ to be sent to you.

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

Changes

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.

perlfaq1 – General Questions About Perl ($Revision: 1.10 $)

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 last production release was 5.003, and the current experimental release for those at the bleeding edge (as of 27/03/97) is 5.003_92, considered a beta for production release 5.004, which will probably be out by the time you read this. Further references to the Perl language in this document refer to the current production release unless otherwise specified.

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.

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?

Perl is easy to start learning -- and easy to keep learning. It looks like most programming languages you're likely to have had experience with, so if you've ever written an C program, an awk script, a shell script, or even an Excel macro, 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 bytecode 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.

perlfaq2 – Obtaining and Learning about Perl ($Revision: 1.13 $)

This section of the FAQ answers questions about where to find source and documentation for Perl, support and training, 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.

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.

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. 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.

It's also worth noting that there's a PDF version of the complete documentation for perl available in the CPAN/authors/id/BMIDD directory.

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.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.
```

There is also USENET gateway to the mailing list used by the crack Perl development team (perl5-porters) at news://genetics.upenn.edu/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.

Perl Books

A number 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 and German translations in progress)
```

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!

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.

If you're already a hard-core systems programmer, then the Camel Book just might suffice for you to learn Perl from. But if you're not, check out the ''Llama Book''. It currently doesn't cover perl5, but the 2nd edition is nearly done and should be out by summer 97:

```
    Learning Perl (the Llama Book):
        Author: Randal Schwartz, with intro by Larry Wall
        ISBN 1-56592-042-2      (English)
        ISBN 4-89502-678-1      (Japanese)
        ISBN 2-84177-005-2      (French)
        ISBN 3-930673-08-8      (German)
```

Another stand-out book in the turquoise O'Reilly Perl line is the "Hip Owls" book. It covers regular expressions inside and out, with quite a bit devoted exclusively to Perl:

```
    Mastering Regular Expressions (the Cute Owls Book):
        Author: Jeffrey Friedl
        ISBN 1-56592-257-3
```

You can order any of these 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. See http://www.ora.com/ on the Web.

Recommended Perl books that are not from O'Reilly are the following:

```
   Cross-Platform Perl, (for Unix and Windows NT)
       Author: Eric F. Johnson
       ISBN: 1-55851-483-X

   How to Set up and Maintain a World Wide Web Site, (2nd edition)
        Author: Lincoln Stein, M.D., Ph.D.
        ISBN: 0-201-63462-7

   CGI Programming in C & Perl,
        Author: Thomas Boutell
        ISBN: 0-201-42219-0
```

Note that some of these address specific application areas (e.g. the Web) and are not general-purpose programming books.

Perl in Magazines

The Perl Journal is the first and only magazine dedicated to Perl. It is published (on paper, not online) quarterly by Jon Orwant (orwant@tpj.com), editor. Subscription information is at http://tpj.com or via email to subscriptions@tpj.com.

Beyond this, two other magazines that frequently carry high-quality articles on Perl are Web Techniques (see http://www.webtechniques.com/) and Unix Review (http://www.unixreview.com/).

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/
```

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 emailing 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, email 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 PerlArchive 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.

Perl Training

While some large training companies offer their own courses on Perl, you may prefer to contact individuals near and dear to the heart of Perl development. Two well-known members of the Perl development team who offer such things are Tom Christiansen <perl-classes@perl.com> and Randal Schwartz <perl-training-info@stonehenge.com>, plus their respective minions, who offer a variety of professional tutorials and seminars on Perl. These courses include large public seminars, private corporate training, and fly-ins to Colorado and Oregon. See http://www.perl.com/perl/info/training.html for more details.

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 email 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 man page (perl5.004 or later) for more information.

What is perl.com? perl.org? The Perl Institute?

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.

The perl.com domain is Tom Christiansen's domain. He created it as a public service long before perl.org came about. It's the original PBS of the Perl world, a clearinghouse for information about all things Perlian, accepting no paid advertisements, glossy gifs, or (gasp!) java applets on its pages.

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.

perlfaq3 – Programming Tools ($Revision: 1.19 $)

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:

```
Objects         perlref, perlmod, perlobj, perltie
Data Structures perlref, perllol, perldsc
Modules         perlmod, perlsub
Regexps         perlre, perlfunc, perlop
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`?

Have you tried `use strict`?

Did you check the returns of each and every system call?

Did you read the *perltrap* manpage?

Have you tried the Perl debugger, described in the *perldebug* manpage?

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.

How do I cross-reference my Perl programs?

The B::Xref module, shipped with the new, alpha-release Perl compiler (not the general distribution), can be used to generate cross-reference reports for Perl programs.

```
perl -MO=Xref[,OPTIONS] foo.pl
```

Is there a pretty-printer (formatter) for Perl?

There is no program that will reformat Perl as much as *indent*(1) will do 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.

Your editor can and should help you with source formatting. 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 using 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 ftp://ftp.perl.com/pub/vi/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", anyway.

How can I use curses with Perl?

The Curses module from CPAN provides a dynamically loadable object module interface to a curses library.

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.

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.

Can I dynamically load C routines into Perl?

If your system architecture supports it, then the standard perl on your system should also provide you with this via the DynaLoader module. Read the *perlxstut* manpage for details.

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.

Other approaches include autoloading 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.

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 built-in 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. Memory the system allocates to a program will never be returned to the system. That's why long-running programs sometimes re-exec themselves.

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. (NB: *my()* variables also execute about 10% faster than globals.) 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 at least 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::* modules (from CPAN), 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. 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 processes.

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.

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.) 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 Feb-1997 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.

Please understand that 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 thus will still 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.

Malcolm will be in charge of the 5.005 release of Perl itself to try to unify and merge his compiler and multithreading work into the main release.

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!

How can I get '#!perl' to work on [MSDOS,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 Activeware port of Perl, will modify the Registry to associate the .pl extension with the perl interpreter. If you install another 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 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, I'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.

I'm afraid that there is no general solution to all of this. It is a mess, pure and simple.

[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:

```
The Idiot's Guide to Solving Perl/CGI Problems, by Tom Christiansen
    http://www.perl.com/perl/faq/idiots-guide.html

Frequently Asked Questions about CGI Programming, by Nick Kew
    ftp://rtfm.mit.edu/pub/usenet/news.answers/www/cgi-faq
    http://www3.pair.com/webthing/docs/cgi/faqs/cgifaq.shtml

Perl/CGI programming FAQ, by Shishir Gundavaram and Tom Christiansen
    http://www.perl.com/perl/faq/perl-cgi-faq.html

The WWW Security FAQ, by Lincoln Stein
    http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html

World Wide Web FAQ, by Thomas Boutell
    http://www.boutell.com/faq/
```

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.

perlfaq4 – Data Manipulation ($Revision: 1.15 $)

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 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?

For rounding to a certain number of digits, *sprintf()* or *printf()* is usually the easiest route.

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. It's not as efficient as the POSIX library, but the POSIX library 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.

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 that's how these things work. A longer explanation is available on http://www.perl.com/CPAN/doc/FMTEYEWTK/random, courtesy of Tom Phoenix.

You should also check out the Math::TrulyRandom module from CPAN.

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.

How can I compare two date strings?

Use the Date::Manip or Date::DateCalc 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 one of the Date modules from CPAN.

How can I find the Julian Day?

Neither Date::Manip nor Date::DateCalc deal with Julian days. Instead, there is an example of Julian date calculation in http://www.perl.com/CPAN/authors/David_Muir_Sharnoff/modules/Time/JulianDay.pm.gz, which should help.

Does Perl have a year 2000 problem?

Not unless you use Perl to create one. 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). 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 a 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.

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, email 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;
```

Note that 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 a 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";
```

How do I find matching/nesting anything?

This isn't something that can be tackled 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.

How do I reverse a string?

Use *reverse()* in a 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 the old-fashioned way:

```
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);
```

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 regexp kind of thought process will likely prefer

```
$a =~ s/^.../Tom/;
```

How do I change the Nth occurrence of something?

You have to keep track. For example, let's say you want to change the fifth occurrence of "whoever" or "whomever" into "whosoever", case insensitively.

```
$count = 0;
s{((whom?)ever)}{
    ++$count == 5           # is it the 5th?
        ? "${2}soever"      # yes, swap
        : $1                # renege and leave it there
}igex;
```

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;

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;

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 ',';
```

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?

The simplest approach, albeit not the fastest, is probably like this:

```
$string =~ s/^\s*(.*?)\s*$/$1/;
```

It would be faster to do this in two steps:

```
$string =~ s/^\s+//;
$string =~ s/\s+$//;
```

Or more nicely written as:

```
for ($string) {
    s/^\s+//;
    s/\s+$//;
}
```

How do I extract selected columns from a string?

Use *substr()* or *unpack()*, both documented in the *perlfunc* manpage.

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';
$text =~ s/\$(\w+)/${$1}/g;
```

Before version 5 of perl, this had to be done with a double-eval substitution:

```
$text =~ s/(\$\w+)/$1/eeg;
```

Which is bizarre enough that you'll probably actually need an EEG afterwards. :–)

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.

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.

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:

```
$prev = 'nonesuch';
@out = grep($_ 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.

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 an array contains a certain element?

There are several ways to approach this. If you are going to make this query many times and the values are arbitrary strings, 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 $read;
grep (vec($read,$_,1) = 1, @articles);
```

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.

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?

Here's a shuffling algorithm which works its way through the list, 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));
}
```

For large arrays, this avoids a lot of the reshuffling:

```
srand;
@new = ();
@old = 1 .. 10000;  # just a demo
for( @old ){
    my $r = rand @new+1;
    push(@new,$new[$r]);
    $new[$r] = $_;
}
```

How do I process/modify each element of an array?

Use `for/foreach`:

```
for (@lines) {
    s/foo/bar/;
    tr[a-z][A-Z];
}
```

Here's another; let's compute spherical volumes:

```
for (@radii) {
    $_ **= 3;
    $_ *= (4/3) * 3.14159;  # this will be constant folded
}
```

How do I select a random element from an array?

Use the *rand()* function (see the `rand` entry in the *perlfunc* manpage):

```
srand;                      # not needed for 5.004 and later
$index   = rand @array;
$element = $array[$index];
```

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 *permut()* function should work on any list:

```
#!/usr/bin/perl -n
# permute - tchrist@perl.com
permut([split], []);
sub permut {
    my @head = @{ $_[0] };
    my @tail = @{ $_[1] };
    unless (@head) {
        # stop recursing when there are no elements in the head
        print "@tail\n";
    } else {
        # for all elements in @head, move one from @head to @tail
        # and call permut() on the new @head and @tail
        my(@newhead,@newtail,$i);
        foreach $i (0 .. $#head) {
            @newhead = @head;
            @newtail = @tail;
            unshift(@newtail, splice(@newhead, $i, 1));
            permut([@newhead], [@newtail]);
        }
    }
}
```

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.

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 a 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 reenter 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.

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 multi-byte 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 whole number"   unless /^\d+$/;
warn "not an integer"       unless /^-?\d+$/;  # reject +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+)$/;
warn "not a C float"
    unless /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/;
```

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.

perlfaq5 – Files and Formats ($Revision: 1.19 $)

This section deals with I/O and the "f" issues: filehandles, flushing, formats, and footers.

How do I flush/unbuffer a 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 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, as in the older:

```
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();
$sock->print("GET /\015\012");
$document = join('', $sock->getlines());
print "DOC IS: $document\n";
```

Note the 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.

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]);
```

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. Replacing a sequence of bytes with another sequence of the same length is one. 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.

```
$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;
```

How do I make a temporary file name?

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 IO::File;
    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 {
        my $fh = undef;
        my $count = 0;
        until (defined($fh) || $count > 100) {
            $base_name =~ s/-(\d+)$/"-" . (1 + $1)/e;
            $fh = IO::File->new($base_name, O_WRONLY|O_EXCL|O_CREAT, 0644)
        }
        if (defined($fh)) {
            return ($fh, $base_name);
        } else {
            return ();
        }
    }
}
```

Or you could simply use IO::Handle::new_tmpfile.

How can I manipulate fixed-record-length files?

The most efficient way is using *pack()* and *unpack()*. This is faster than using *substr()*. 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|");
$_ = <PS>; print;
while (<PS>) {
    ($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";
}
```

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?

You may have some success with typeglobs, as we always had to use in days of old:

```
local(*FH);
```

But while still supported, that isn't the best to go about getting local filehandles. Typeglobs have their drawbacks. You may well want to use the `FileHandle` module, which creates new filehandles for you (see the *FileHandle* manpage):

```
use FileHandle;
sub findme {
    my $fh = FileHandle->new();
    open($fh, "</etc/hosts") or die "no /etc/hosts: $!";
    while (<$fh>) {
        print if /\b127\.(0\.0\.)?1\b/;
    }
    # $fh automatically closes/disappears here
}
```

Internally, Perl believes filehandles to be of class IO::Handle. You may use that module directly if you'd like (see the *IO::Handle* manpage), or one of its more specific derived classes.

How can I set up a footer format to be used with *write()*?

There's no built-in 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.

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 the 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
```

Whoops. You should instead use this, which will fail if the file doesn't exist.

```
open(FH, "+< /path/name");  # open for update
```

If this is an issue, try:

```
sysopen(FH, "/path/name", O_RDWR|O_CREAT, 0644);
```

Error checking is left as an exercise for the reader.

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.

How can I lock a file?

Perl's built-in *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.

   The CPAN module File::Lock offers similar functionality and (if you have dynamic loading) won't require you to rebuild perl if your *flock()* can't lock network files.

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, 0644)
            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?

Anyway, this is what to do:

```
use Fcntl;
sysopen(FH, "numfile", O_RDWR|O_CREAT, 0644) 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.

Don't forget to set *binmode()* under DOS–like platforms when operating on files that have anything other than straight text in them. See the docs on *open()* and on *binmode()* for more details.

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];
print "file $file updated at ", scalar(localtime($file)), "\n";
```

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");
```

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.

In theory a IO::Tee class could be written, but to date we haven't seen such.

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?

You should 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);
}
```

You should look into getting the Term::ReadKey extension from CPAN.

How do I open a file without blocking?

You need 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 create a file only if it doesn't exist?

You need to use the O_CREAT and O_EXCL flags from the Fcntl module in conjunction with *sysopen()*:

```
use Fcntl;
sysopen(FH, "/tmp/somefile", O_WRONLY|O_EXCL|O_CREAT, 0644)
            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!

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)
```

Error checking 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.

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 Every 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.

perlfaq6 – Regexps ($Revision: 1.14 $)

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) /ge;
```

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 newlines in your string, or you aren't using the correct *modifier*(s) on your pattern.

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+)(\s+\1)+\b/gi ) {
        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.

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

```
undef $/;
@records = split /your_pattern/, <FH>;
```

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 accented characters?

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+|.[^/"'\\]*)#$2#g;
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:

```
while (<>) {
  chomp;
  PARSER: {
       m/ \G( \d+\b    )/gx     && do { print "number: $1\n";  redo; };
       m/ \G( \w+      )/gx     && do { print "word:   $1\n";  redo; };
       m/ \G( \s+      )/gx     && do { print "space:  $1\n";  redo; };
       m/ \G( [^\w\d]+ )/gx     && do { print "other:  $1\n";  redo; };
  }
}
```

Of course, that could have been written as

```
while (<>) {
  chomp;
  PARSER: {
       if ( /\G( \d+\b    )/gx  {
            print "number: $1\n";
            redo PARSER;
       }
       if ( /\G( \w+      )/gx  {
            print "word: $1\n";
            redo PARSER;
       }
       if ( /\G( \s+      )/gx  {
            print "space: $1\n";
            redo PARSER;
       }
       if ( /\G( [^\w\d]+ )/gx  {
            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?

Strictly speaking, nothing. Stylistically speaking, it's not a good way to write maintainable code. That's because you're using these constructs not for their return values but rather for their side-effects, and side-effects can be mystifying. There's no void *grep()* that's not better written as a `for` (well, `foreach`, technically) loop.

How can I match strings with multi-byte 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 ``maritan'' 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.

perlfaq7 – Perl Language Issues ($Revision: 1.15 $)

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?

No, 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.

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: 1.15 $=~/\d+/g;sprintf '%d.'.'%02d'x$#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.

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 create filehandles you can pass to subroutines, you can use `*FH` or `\*FH` notation ("typeglobs" – see the *perldata* manpage for more information), or create filehandles dynamically using the old FileHandle or the new IO::File modules, both part of the standard Perl distribution.

```
use Fcntl;
use IO::File;
my $fh = new IO::File $filename, O_WRONLY|O_APPEND;
            or die "Can't append to $filename: $!";
func($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.

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 also the *perlsub* manpage, which explains this all in more detail.

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 "*local* ($foo) = <FILE>;" work right?

`local()` gives 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 built-in 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.

Here's a simple example of a switch based on pattern matching. 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";

    }
```

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] ) {

            /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.

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 = ref bless [];
```

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";
}
```

perlfaq8 – System Interaction ($Revision: 1.15 $)

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 $^O variable ($OSTYPE 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
```

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.

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, as in the older

```
use FileHandle;
DEV->autoflush(1);
```

and the newer

```
use IO::Handle;
DEV->autoflush(1);
```

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]);
```

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.

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. 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 `opening` 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 you then trap. 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, the getpw*() functions described in the *perlfunc* manpage provide (read-only) access to 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";
```

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 you 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). It 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).

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 .

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 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 (eg, 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.

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.

Why can't my script read from STDIN after I gave it EOF (^D on Unix, ^Z on MSDOS)?

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 and TCP::Client modules (available from CPAN). http://www.perl.com/CPAN/scripts/netstuff/telnet.emul.shar will also help for emulating the telnet protocol.

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. These days, your best bet is to look at the Comm.pl library available from CPAN.

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.

VMS Change to `%ENV` persist after Perl exits, but directory changes do not.

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.
- 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 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 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 .

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.

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.

perlfaq9 – Networking ($Revision: 1.13 $)

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. Can you help me fix it?

Sure, but you probably can't afford our contracting rates :–)

Seriously, 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 are:

```
http://www.perl.com/perl/faq/idiots-guide.html
http://www3.pair.com/webthing/docs/cgi/faqs/cgifaq.shtml
http://www.perl.com/perl/faq/perl-cgi-faq.html
http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html
http://www.boutell.com/faq/
```

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

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, 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?

Use the LWP::Simple module available from CPAN, part of the excellent libwww-perl (LWP) package. On the other hand, and if you have the lynx text-based HTML browser installed on your system, this isn't too bad:

```
$html_code = `lynx -source $url`;
$text_data = `lynx -dump $url`;
```

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=%2Bcgi-bin+%2Bperl.exe";
$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.

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 parse an email 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?

A lot of people are tempted to code this up themselves, so you’ve probably all seen a lot of code involving `$ENV{CONTENT_LENGTH}` and `$ENV{QUERY_STRING}`. It’s true that this can work, but there are also a lot of versions of this floating around that are quite simply broken!

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).

How do I check a valid email address?

You can’t.

Without sending mail to the address and seeing whether it bounces (and even then you face the halting problem), you cannot determine whether an email address is valid. Even if you apply the email 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 email addresses with a simple regexp, such as

`/^[\w.-]+\@([\w.-]\.)+\w+$/`. 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 email to (say, Bill Clinton or your postmaster), and then makes sure that the hostname given can be looked up in DNS. It's not fast, but it works.

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 email 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 email address can mean that this generates addresses that the company's email system will not accept, so you should ask for users' email 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/read mail?

Sending mail: the Mail::Mailer module from CPAN (part of the MailTools package) is UNIX−centric, while Mail::Internet uses Net::SMTP which is not UNIX−centric. Reading 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).

How do I find out my hostname/domainname/IP address?

A lot of code has historically cavalierly called the `` `hostname` `` program. While sometimes expedient, this isn't very portable. 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.