NAME

perl − The Perl 5 language interpreter

SYNOPSIS

perl [**−sTtuUWX**] [**−hv**] [**−V**[:*configvar*]]
[**−cw**] [**−d**[**t**][:*debugger*]] [**−D**[*number/list*]]
[**−pna**] [**−F***pattern*] [**−l**[*octal*]] [**−0**[*octal/hexadecimal*]]
[**−I***dir*] [**−m**[**−**]*module*] [**−M**[**−**]*'module...'*] [**−f**] [**−C [*number/list*]**] [**−S**]
[**−x**[*dir*]] [**−i**[*extension*]]
[[**−e**|**−E**] *'command'*] [**−−**] [*programfile*] [*argument*]...

For more information on these options, you can run `perldoc perlrun`.

GETTING HELP

The *perldoc* program gives you access to all the documentation that comes with Perl. You can get more documentation, tutorials and community support online at <http://www.perl.org/>.

If you're new to Perl, you should start by running `perldoc perlintro`, which is a general intro for beginners and provides some background to help you navigate the rest of Perl's extensive documentation. Run `perldoc perldoc` to learn more things you can do with *perldoc*.

For ease of access, the Perl manual has been split up into several sections.

Overview

```
perl                Perl overview (this section)
perlintro           Perl introduction for beginners
perlrun             Perl execution and options
perltoc             Perl documentation table of contents
```

Tutorials

```
perlreftut          Perl references short introduction
perldsc             Perl data structures intro
perllol             Perl data structures: arrays of arrays

perlrequick         Perl regular expressions quick start
perlretut           Perl regular expressions tutorial

perlootut           Perl OO tutorial for beginners

perlperf            Perl Performance and Optimization Techniques

perlstyle           Perl style guide

perlcheat           Perl cheat sheet
perltrap            Perl traps for the unwary
perldebtut          Perl debugging tutorial

perlfaq             Perl frequently asked questions
  perlfaq1          General Questions About Perl
  perlfaq2          Obtaining and Learning about Perl
  perlfaq3          Programming Tools
  perlfaq4          Data Manipulation
  perlfaq5          Files and Formats
  perlfaq6          Regexes
  perlfaq7          Perl Language Issues
  perlfaq8          System Interaction
  perlfaq9          Networking
```

Reference Manual

perlsyn	Perl syntax
perldata	Perl data structures
perlop	Perl operators and precedence
perlsub	Perl subroutines
perlfunc	Perl built-in functions
perlopentut	Perl open() tutorial
perlpacktut	Perl pack() and unpack() tutorial
perlpod	Perl plain old documentation
perlpodspec	Perl plain old documentation format specification
perlpodstyle	Perl POD style guide
perldiag	Perl diagnostic messages
perldeprecation	Perl deprecations
perllexwarn	Perl warnings and their control
perldebug	Perl debugging
perlvar	Perl predefined variables
perlre	Perl regular expressions, the rest of the story
perlrebackslash	Perl regular expression backslash sequences
perlrecharclass	Perl regular expression character classes
perlreref	Perl regular expressions quick reference
perlref	Perl references, the rest of the story
perlform	Perl formats
perlobj	Perl objects
perltie	Perl objects hidden behind simple variables
perldbmfilter	Perl DBM filters
perlipc	Perl interprocess communication
perlfork	Perl fork() information
perlnumber	Perl number semantics
perlthrtut	Perl threads tutorial
perlport	Perl portability guide
perllocale	Perl locale support
perluniintro	Perl Unicode introduction
perlunicode	Perl Unicode support
perlunicook	Perl Unicode cookbook
perlunifaq	Perl Unicode FAQ
perluniprops	Index of Unicode properties in Perl
perlunitut	Perl Unicode tutorial
perlebcdic	Considerations for running Perl on EBCDIC platforms
perlsec	Perl security
perlmod	Perl modules: how they work
perlmodlib	Perl modules: how to write and use
perlmodstyle	Perl modules: how to write modules with style
perlmodinstall	Perl modules: how to install from CPAN
perlnewmod	Perl modules: preparing a new module for distribution
perlpragma	Perl modules: writing a user pragma
perlutil	utilities packaged with the Perl distribution
perlfilter	Perl source filters
perldtrace	Perl's support for DTrace
perlglossary	Perl Glossary

Internals and C Language Interface

perlembed	Perl ways to embed perl in your C or C++ application
perldebguts	Perl debugging guts and tips
perlxstut	Perl XS tutorial
perlxs	Perl XS application programming interface
perlxstypemap	Perl XS C/Perl type conversion tools
perlclib	Internal replacements for standard C library functions
perlguts	Perl internal functions for those doing extensions
perlcall	Perl calling conventions from C
perlmroapi	Perl method resolution plugin interface
perlreapi	Perl regular expression plugin interface
perlreguts	Perl regular expression engine internals
perlapi	Perl API listing (autogenerated)
perlintern	Perl internal functions (autogenerated)
perliol	C API for Perl's implementation of IO in Layers
perlapio	Perl internal IO abstraction interface
perlhack	Perl hackers guide
perlsource	Guide to the Perl source tree
perlinterp	Overview of the Perl interpreter source and how it works
perlhacktut	Walk through the creation of a simple C code patch
perlhacktips	Tips for Perl core C code hacking
perlpolicy	Perl development policies
perlgit	Using git with the Perl repository

Miscellaneous

perlbook	Perl book information
perlcommunity	Perl community information
perldoc	Look up Perl documentation in Pod format
perlhist	Perl history records
perldelta	Perl changes since previous version
perl5260delta	Perl changes in version 5.26.0
perl5242delta	Perl changes in version 5.24.2
perl5241delta	Perl changes in version 5.24.1
perl5240delta	Perl changes in version 5.24.0
perl5224delta	Perl changes in version 5.22.4
perl5223delta	Perl changes in version 5.22.3
perl5222delta	Perl changes in version 5.22.2
perl5221delta	Perl changes in version 5.22.1
perl5220delta	Perl changes in version 5.22.0
perl5203delta	Perl changes in version 5.20.3
perl5202delta	Perl changes in version 5.20.2
perl5201delta	Perl changes in version 5.20.1
perl5200delta	Perl changes in version 5.20.0
perl5184delta	Perl changes in version 5.18.4
perl5182delta	Perl changes in version 5.18.2
perl5181delta	Perl changes in version 5.18.1
perl5180delta	Perl changes in version 5.18.0
perl5163delta	Perl changes in version 5.16.3
perl5162delta	Perl changes in version 5.16.2
perl5161delta	Perl changes in version 5.16.1
perl5160delta	Perl changes in version 5.16.0
perl5144delta	Perl changes in version 5.14.4
perl5143delta	Perl changes in version 5.14.3
perl5142delta	Perl changes in version 5.14.2
perl5141delta	Perl changes in version 5.14.1

perl5140delta Perl changes in version 5.14.0
perl5125delta Perl changes in version 5.12.5
perl5124delta Perl changes in version 5.12.4
perl5123delta Perl changes in version 5.12.3
perl5122delta Perl changes in version 5.12.2
perl5121delta Perl changes in version 5.12.1
perl5120delta Perl changes in version 5.12.0
perl5101delta Perl changes in version 5.10.1
perl5100delta Perl changes in version 5.10.0
perl589delta Perl changes in version 5.8.9
perl588delta Perl changes in version 5.8.8
perl587delta Perl changes in version 5.8.7
perl586delta Perl changes in version 5.8.6
perl585delta Perl changes in version 5.8.5
perl584delta Perl changes in version 5.8.4
perl583delta Perl changes in version 5.8.3
perl582delta Perl changes in version 5.8.2
perl581delta Perl changes in version 5.8.1
perl58delta Perl changes in version 5.8.0
perl561delta Perl changes in version 5.6.1
perl56delta Perl changes in version 5.6
perl5005delta Perl changes in version 5.005
perl5004delta Perl changes in version 5.004

perlexperiment A listing of experimental features in Perl

perlartistic Perl Artistic License
perlgpl GNU General Public License

Language-Specific

perlcn Perl for Simplified Chinese (in EUC-CN)
perljp Perl for Japanese (in EUC-JP)
perlko Perl for Korean (in EUC-KR)
perltw Perl for Traditional Chinese (in Big5)

Platform-Specific

perlaix Perl notes for AIX
perlamiga Perl notes for AmigaOS
perlandroid Perl notes for Android
perlbs2000 Perl notes for POSIX-BC BS2000
perlce Perl notes for WinCE
perlcygwin Perl notes for Cygwin
perldos Perl notes for DOS
perlfreebsd Perl notes for FreeBSD
perlhaiku Perl notes for Haiku
perlhpux Perl notes for HP-UX
perlhurd Perl notes for Hurd
perlirix Perl notes for Irix
perllinux Perl notes for Linux
perlmacos Perl notes for Mac OS (Classic)
perlmacosx Perl notes for Mac OS X
perlnetware Perl notes for NetWare
perlopenbsd Perl notes for OpenBSD
perlos2 Perl notes for OS/2
perlos390 Perl notes for OS/390
perlos400 Perl notes for OS/400
perlplan9 Perl notes for Plan 9
perlqnx Perl notes for QNX
perlriscos Perl notes for RISC OS
perlsolaris Perl notes for Solaris

```
perlsymbian         Perl notes for Symbian
perlsynology        Perl notes for Synology
perltru64           Perl notes for Tru64
perlvms             Perl notes for VMS
perlvos             Perl notes for Stratus VOS
perlwin32           Perl notes for Windows
```

Stubs for Deleted Documents

```
perlboot
perlbot
perlrepository
perltodo
perltooc
perltoot
```

On Debian systems, you need to install the **perl-doc** package which contains the majority of the standard Perl documentation and the *perldoc* program.

Extensive additional documentation for Perl modules is available, both those distributed with Perl and third-party modules which are packaged or locally installed.

You should be able to view Perl's documentation with your *man* (1) program or *perldoc* (1).

Some documentation is not available as man pages, so if a cross-reference is not found by man, try it with perldoc. Perldoc can also take you directly to documentation for functions (with the **−f** switch). See `perldoc --help` (or `perldoc perldoc` or `man perldoc`) for other helpful options perldoc has to offer.

In general, if something strange has gone wrong with your program and you're not sure where you should look for help, try making your code comply with **use strict** and **use warnings**. These will often point out exactly where the trouble is.

DESCRIPTION

Perl officially stands for Practical Extraction and Report Language, except when it doesn't.

Perl was originally a language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It quickly became a good language for many system management tasks. Over the years, Perl has grown into a general-purpose programming language. It's widely used for everything from quick ''one-liners'' to full-scale application development.

The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). It combines (in the author's opinion, anyway) some of the best features of **sed**, **awk**, and **sh**, making it familiar and easy to use for Unix users to whip up quick solutions to annoying problems. Its general-purpose programming facilities support procedural, functional, and object-oriented programming paradigms, making Perl a comfortable language for the long haul on major projects, whatever your bent.

Perl's roots in text processing haven't been forgotten over the years. It still boasts some of the most powerful regular expressions to be found anywhere, and its support for Unicode text is world-class. It handles all kinds of structured text, too, through an extensive collection of extensions. Those libraries, collected in the CPAN, provide ready-made solutions to an astounding array of problems. When they haven't set the standard themselves, they steal from the best — just like Perl itself.

AVAILABILITY

Perl is available for most operating systems, including virtually all Unix-like platforms. See ''Supported Platforms'' in perlport for a listing.

ENVIRONMENT

See perlrun.

AUTHOR

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

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

FILES

`"@INC"`	`locations of perl libraries`

"@INC" above is a reference to the built-in variable of the same name; see perlvar for more information.

SEE ALSO

`http://www.perl.org/`	`the Perl homepage`
`http://www.perl.com/`	`Perl articles (O'Reilly)`
`http://www.cpan.org/`	`the Comprehensive Perl Archive`
`http://www.pm.org/`	`the Perl Mongers`

DIAGNOSTICS

Using the `use strict` pragma ensures that all variables are properly declared and prevents other misuses of legacy Perl features.

The `use warnings` pragma produces some lovely diagnostics. One can also use the **−w** flag, but its use is normally discouraged, because it gets applied to all executed Perl code, including that not under your control.

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

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

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

Did we mention that you should definitely consider using the **use warnings** pragma?

BUGS

The behavior implied by the **use warnings** pragma is not mandatory.

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

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

While none of the built-in data types have any arbitrary size limits (apart from memory size), there are still a few arbitrary limits: a given variable name may not be longer than 251 characters. Line numbers displayed by diagnostics are internally stored as short integers, so they are limited to a maximum of 65535 (higher numbers usually being affected by wraparound).

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

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

NOTES

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

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

NAME

perlsyn – Perl syntax

DESCRIPTION

A Perl program consists of a sequence of declarations and statements which run from the top to the bottom. Loops, subroutines, and other control structures allow you to jump around within the code.

Perl is a **free-form** language: you can format and indent it however you like. Whitespace serves mostly to separate tokens, unlike languages like Python where it is an important part of the syntax, or Fortran where it is immaterial.

Many of Perl's syntactic elements are **optional**. Rather than requiring you to put parentheses around every function call and declare every variable, you can often leave such explicit elements off and Perl will figure out what you meant. This is known as **Do What I Mean**, abbreviated **DWIM**. It allows programmers to be **lazy** and to code in a style with which they are comfortable.

Perl **borrows syntax** and concepts from many languages: awk, sed, C, Bourne Shell, Smalltalk, Lisp and even English. Other languages have borrowed syntax from Perl, particularly its regular expression extensions. So if you have programmed in another language you will see familiar pieces in Perl. They often work the same, but see perltrap for information about how they differ.

Declarations

The only things you need to declare in Perl are report formats and subroutines (and sometimes not even subroutines). A scalar variable holds the undefined value (`undef`) until it has been assigned a defined value, which is anything other than `undef`. When used as a number, `undef` is treated as `0`; when used as a string, it is treated as the empty string, `""`; and when used as a reference that isn't being assigned to, it is treated as an error. If you enable warnings, you'll be notified of an uninitialized value whenever you treat `undef` as a string or a number. Well, usually. Boolean contexts, such as:

```
if ($a) {}
```

are exempt from warnings (because they care about truth rather than definedness). Operators such as `++`, `--`, `+=`, `-=`, and `.=`, that operate on undefined variables such as:

```
undef $a;
$a++;
```

are also always exempt from such warnings.

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

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

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

A bare declaration like that declares the function to be a list operator, not a unary operator, so you have to be careful to use parentheses (or `or` instead of `||`.) The `||` operator binds too tightly to use after list operators; it becomes part of the last element. You can always use parentheses around the list operators arguments to turn the list operator back into something that behaves more like a function call. Alternatively, you can use the prototype `($)` to turn the subroutine into a unary operator:

```
sub myname ($);
$me = myname $0             || die "can't get myname";
```

That now parses as you'd expect, but you still ought to get in the habit of using parentheses in that situation. For more on prototypes, see perlsub.

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

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

effects.

Comments

Text from a "#" character until the end of the line is a comment, and is ignored. Exceptions include "#" inside a string or regular expression.

Simple Statements

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

Truth and Falsehood

The number 0, the strings `'0'` and `""`, the empty list `()`, and `undef` are all false in a boolean context. All other values are true. Negation of a true value by `!` or `not` returns a special false value. When evaluated as a string it is treated as `""`, but as a number, it is treated as 0. Most Perl operators that return true or false behave this way.

Statement Modifiers

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

```
if EXPR
unless EXPR
while EXPR
until EXPR
for LIST
foreach LIST
when EXPR
```

The `EXPR` following the modifier is referred to as the "condition". Its truth or falsehood determines how the modifier will behave.

`if` executes the statement once *if* and only if the condition is true. `unless` is the opposite, it executes the statement *unless* the condition is true (that is, if the condition is false).

```
print "Basset hounds got long ears" if length $ear >= 10;
go_outside() and play() unless $is_raining;
```

The `for(each)` modifier is an iterator: it executes the statement once for each item in the LIST (with `$_` aliased to each item in turn).

```
print "Hello $_!\n" for qw(world Dolly nurse);
```

`while` repeats the statement *while* the condition is true. `until` does the opposite, it repeats the statement *until* the condition is true (or while the condition is false):

```
# Both of these count from 0 to 10.
print $i++ while $i <= 10;
print $j++ until $j >  10;
```

The `while` and `until` modifiers have the usual "`while` loop" semantics (conditional evaluated first), except when applied to a `do`-BLOCK (or to the Perl4 `do`-SUBROUTINE statement), in which case the block executes once before the conditional is evaluated.

This is so that you can write loops like:

```
do {
    $line = <STDIN>;
    ...
} until !defined($line) || $line eq ".\n"
```

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

For `next` or `redo`, just double the braces:

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

For `last`, you have to be more elaborate and put braces around it:

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

If you need both `next` and `last`, you have to do both and also use a loop label:

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

NOTE: The behaviour of a `my`, `state`, or `our` modified with a statement modifier conditional or loop construct (for example, `my $x if ...`) is **undefined**. The value of the `my` variable may be `undef`, any previously assigned value, or possibly anything else. Don't rely on it. Future versions of perl might do something different from the version of perl you try it out on. Here be dragons.

The `when` modifier is an experimental feature that first appeared in Perl 5.14. To use it, you should include a `use v5.14` declaration. (Technically, it requires only the `switch` feature, but that aspect of it was not available before 5.14.) Operative only from within a `foreach` loop or a `given` block, it executes the statement only if the smartmatch `$_ ~~ `*EXPR* is true. If the statement executes, it is followed by a `next` from inside a `foreach` and `break` from inside a `given`.

Under the current implementation, the `foreach` loop can be anywhere within the `when` modifier's dynamic scope, but must be within the `given` block's lexical scope. This restriction may be relaxed in a future release. See "Switch Statements" below.

Compound Statements

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

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

The following compound statements may be used to control flow:

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

unless (EXPR) BLOCK
unless (EXPR) BLOCK else BLOCK
unless (EXPR) BLOCK elsif (EXPR) BLOCK ...
unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

given (EXPR) BLOCK

LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK

LABEL until (EXPR) BLOCK
```

```
LABEL until (EXPR) BLOCK continue BLOCK

LABEL for (EXPR; EXPR; EXPR) BLOCK
LABEL for VAR (LIST) BLOCK
LABEL for VAR (LIST) BLOCK continue BLOCK

LABEL foreach (EXPR; EXPR; EXPR) BLOCK
LABEL foreach VAR (LIST) BLOCK
LABEL foreach VAR (LIST) BLOCK continue BLOCK

LABEL BLOCK
LABEL BLOCK continue BLOCK

PHASE BLOCK
```

The experimental `given` statement is *not automatically enabled*; see "Switch Statements" below for how to do so, and the attendant caveats.

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

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

The `if` statement is straightforward. Because BLOCKs are always bounded by curly brackets, there is never any ambiguity about which `if` an `else` goes with. If you use `unless` in place of `if`, the sense of the test is reversed. Like `if`, `unless` can be followed by `else`. `unless` can even be followed by one or more `elsif` statements, though you may want to think twice before using that particular language construct, as everyone reading your code will have to think at least twice before they can understand what's going on.

The `while` statement executes the block as long as the expression is true. The `until` statement executes the block as long as the expression is false. The LABEL is optional, and if present, consists of an identifier followed by a colon. The LABEL identifies the loop for the loop control statements `next`, `last`, and `redo`. If the LABEL is omitted, the loop control statement refers to the innermost enclosing loop. This may include dynamically looking back your call-stack at run time to find the LABEL. Such desperate behavior triggers a warning if you use the `use warnings` pragma or the **-w** flag.

If there is a `continue` BLOCK, it is always executed just before the conditional is about to be evaluated again. Thus it can be used to increment a loop variable, even when the loop has been continued via the `next` statement.

When a block is preceding by a compilation phase keyword such as `BEGIN`, `END`, `INIT`, `CHECK`, or `UNITCHECK`, then the block will run only during the corresponding phase of execution. See perlmod for more details.

Extension modules can also hook into the Perl parser to define new kinds of compound statements. These are introduced by a keyword which the extension recognizes, and the syntax following the keyword is defined entirely by the extension. If you are an implementor, see "PL_keyword_plugin" in perlapi for the mechanism. If you are using such a module, see the module's documentation for details of the syntax that it defines.

Loop Control

The `next` command starts the next iteration of the loop:

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

The `last` command immediately exits the loop in question. The `continue` block, if any, is not

executed:

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

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

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

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

which is Perl shorthand for the more explicitly written version:

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

Note that if there were a `continue` block on the above code, it would get executed only on lines discarded by the regex (since redo skips the continue block). A continue block is often used to reset line counters or `m?pat?` one-time matches:

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

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

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

```
if (/pattern/) {{
    last if /fred/;
    next if /barney/; # same effect as "last",
                      # but doesn't document as well
    # do something here
}}
```

This is caused by the fact that a block by itself acts as a loop that executes once, see "Basic BLOCKs".

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

For Loops

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

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

is the same as this:

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

There is one minor difference: if variables are declared with `my` in the initialization section of the `for`, the lexical scope of those variables is exactly the `for` loop (the body of the loop and the control sections).

As a special case, if the test in the `for` loop (or the corresponding `while` loop) is empty, it is treated as true. That is, both

```
for (;;) {
    ...
}
```

and

```
while () {
    ...
}
```

are treated as infinite loops.

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

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

Using `readline` (or the operator form, `<EXPR>`) as the conditional of a `for` loop is shorthand for the following. This behaviour is the same as a `while` loop conditional.

```
for ( prompt(); defined( $_ = <STDIN> ); prompt() ) {
    # do something
}
```

Foreach Loops

The `foreach` loop iterates over a normal list value and sets the scalar variable VAR to be each element of the list in turn. If the variable is preceded with the keyword `my`, then it is lexically scoped, and is therefore visible only within the loop. Otherwise, the variable is implicitly local to the loop and regains its former value upon exiting the loop. If the variable was previously declared with `my`, it uses that variable instead of the global one, but it's still localized to the loop. This implicit localization occurs *only* in a `foreach` loop.

The `foreach` keyword is actually a synonym for the `for` keyword, so you can use either. If VAR is omitted, `$_` is set to each value.

If any element of LIST is an lvalue, you can modify it by modifying VAR inside the loop. Conversely, if any element of LIST is NOT an lvalue, any attempt to modify that element will fail. In other words, the `foreach` loop index variable is an implicit alias for each item in the list that you're looping over.

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

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

either.

As of Perl 5.22, there is an experimental variant of this loop that accepts a variable preceded by a backslash for VAR, in which case the items in the LIST must be references. The backslashed variable will become an alias to each referenced item in the LIST, which must be of the correct type. The variable needn't be a scalar in this case, and the backslash may be followed by `my`. To use this form, you must enable the `refaliasing` feature via `use feature`. (See feature. See also "Assigning to References" in perlref.)

Examples:

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

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

for $count (reverse(1..10), "BOOM") {
    print $count, "\n";
    sleep(1);
}

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

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

use feature "refaliasing";
no warnings "experimental::refaliasing";
foreach \my %hash (@array_of_hash_references) {
    # do something which each %hash
}
```

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

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

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

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

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

Perceptive Perl hackers may have noticed that a `for` loop has a return value, and that this value can be captured by wrapping the loop in a `do` block. The reward for this discovery is this cautionary advice: The return value of a `for` loop is unspecified and may change without notice. Do not rely on it.

Basic BLOCKs

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

The BLOCK construct can be used to emulate case structures.

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

You'll also find that `foreach` loop used to create a topicalizer and a switch:

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

Such constructs are quite frequently used, both because older versions of Perl had no official `switch` statement, and also because the new version described immediately below remains experimental and can sometimes be confusing.

Switch Statements

Starting from Perl 5.10.1 (well, 5.10.0, but it didn't work right), you can say

```
use feature "switch";
```

to enable an experimental switch feature. This is loosely based on an old version of a Perl 6 proposal, but it no longer resembles the Perl 6 construct. You also get the switch feature whenever you declare that your code prefers to run under a version of Perl that is 5.10 or later. For example:

```
use v5.14;
```

Under the "switch" feature, Perl gains the experimental keywords `given`, `when`, `default`, `continue`, and `break`. Starting from Perl 5.16, one can prefix the switch keywords with `CORE::` to access the feature without a `use feature` statement. The keywords `given` and `when` are analogous to `switch` and `case` in other languages — though `continue` is not — so the code in the previous section could be rewritten as

```
use v5.10.1;
for ($var) {
    when (/^abc/) { $abc = 1 }
    when (/^def/) { $def = 1 }
    when (/^xyz/) { $xyz = 1 }
    default       { $nothing = 1 }
}
```

The `foreach` is the non-experimental way to set a topicalizer. If you wish to use the highly experimental `given`, that could be written like this:

```
use v5.10.1;
given ($var) {
    when (/^abc/) { $abc = 1 }
    when (/^def/) { $def = 1 }
    when (/^xyz/) { $xyz = 1 }
    default       { $nothing = 1 }
}
```

As of 5.14, that can also be written this way:

```
use v5.14;
for ($var) {
    $abc = 1 when /^abc/;
    $def = 1 when /^def/;
    $xyz = 1 when /^xyz/;
    default { $nothing = 1 }
}
```

Or if you don't care to play it safe, like this:

```
use v5.14;
given ($var) {
    $abc = 1 when /^abc/;
    $def = 1 when /^def/;
    $xyz = 1 when /^xyz/;
    default { $nothing = 1 }
}
```

The arguments to `given` and `when` are in scalar context, and `given` assigns the `$_` variable its topic value.

Exactly what the *EXPR* argument to `when` does is hard to describe precisely, but in general, it tries to guess what you want done. Sometimes it is interpreted as `$_ ~~ EXPR`, and sometimes it is not. It also behaves differently when lexically enclosed by a `given` block than it does when dynamically enclosed by a `foreach` loop. The rules are far too difficult to understand to be described here. See "Experimental Details on given and when" later on.

Due to an unfortunate bug in how `given` was implemented between Perl 5.10 and 5.16, under those implementations the version of `$_` governed by `given` is merely a lexically scoped copy of the original, not a dynamically scoped alias to the original, as it would be if it were a `foreach` or under both the original and the current Perl 6 language specification. This bug was fixed in Perl 5.18 (and lexicalized `$_` itself was removed in Perl 5.24).

If your code still needs to run on older versions, stick to `foreach` for your topicalizer and you will be less unhappy.

Goto

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

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

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

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

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

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

The Ellipsis Statement

Beginning in Perl 5.12, Perl accepts an ellipsis, "`...`", as a placeholder for code that you haven't implemented yet. This form of ellipsis, the unimplemented statement, should not be confused with the binary flip-flop `...` operator. One is a statement and the other an operator. (Perl doesn't usually confuse them because usually Perl can tell whether it wants an operator or a statement, but see below for exceptions.)

When Perl 5.12 or later encounters an ellipsis statement, it parses this without error, but if and when you should actually try to execute it, Perl throws an exception with the text `Unimplemented`:

```
use v5.12;
sub unimplemented { ... }
eval { unimplemented() };
if ($@ =~ /^Unimplemented at /) {
    say "I found an ellipsis!";
}
```

You can only use the elliptical statement to stand in for a complete statement. These examples of how the ellipsis works:

```
use v5.12;
{ ... }
sub foo { ... }
...;
eval { ... };
sub somemeth {
    my $self = shift;
    ...;
}
$x = do {
    my $n;
    ...;
    say "Hurrah!";
    $n;
};
```

The elliptical statement cannot stand in for an expression that is part of a larger statement, since the `...` is also the three-dot version of the flip-flop operator (see "Range Operators" in perlop).

These examples of attempts to use an ellipsis are syntax errors:

```
use v5.12;

print ...;
open(my $fh, ">", "/dev/passwd") or ...;
if ($condition && ... ) { say "Howdy" };
```

There are some cases where Perl can't immediately tell the difference between an expression and a statement. For instance, the syntax for a block and an anonymous hash reference constructor look the same unless there's something in the braces to give Perl a hint. The ellipsis is a syntax error if Perl doesn't guess that the `{ ... }` is a block. In that case, it doesn't think the `...` is an ellipsis because it's expecting an expression instead of a statement:

```
@transformed = map { ... } @input;     # syntax error
```

Inside your block, you can use a `;` before the ellipsis to denote that the `{ ... }` is a block and not a hash reference constructor. Now the ellipsis works:

```
@transformed = map {; ... } @input;    # ';' disambiguates
```

Note: Some folks colloquially refer to this bit of punctuation as a "yada-yada" or "triple-dot", but its true name is actually an ellipsis.

PODs: Embedded Documentation

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

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

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

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

```
=item snazzle($)

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

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

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

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

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

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

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

Plain Old Comments (Not!)

Perl can process line directives, much like the C preprocessor. Using this, one can control Perl's idea of filenames and line numbers in error or warning messages (especially for strings that are processed with `eval()`). The syntax for this mechanism is almost the same as for most C preprocessors: it matches the regular expression

```
# example: '# line 42 "new_filename.plx"'
/^\#   \s*
  line \s+ (\d+)   \s*
  (?:\s("?)([^"]+)\g2)? \s*
 $/x
```

with `$1` being the line number for the next line, and `$3` being the optional filename (specified with or without quotes). Note that no whitespace may precede the `#`, unlike modern C preprocessors.

There is a fairly obvious gotcha included with the line directive: Debuggers and profilers will only show the last source line to appear at a particular line number in a given file. Care should be taken not to cause line number collisions in code you'd like to debug later.

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

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

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

```
__END__
foo at - line 2001.

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

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

Experimental Details on given and when

As previously mentioned, the "switch" feature is considered highly experimental; it is subject to change with little notice. In particular, `when` has tricky behaviours that are expected to change to become less tricky in the future. Do not rely upon its current (mis)implementation. Before Perl 5.18, `given` also had tricky behaviours that you should still beware of if your code must run on older versions of Perl.

Here is a longer example of `given`:

```
use feature ":5.10";
given ($foo) {
    when (undef) {
        say '$foo is undefined';
    }
    when ("foo") {
        say '$foo is the string "foo"';
    }
    when ([1,3,5,7,9]) {
        say '$foo is an odd digit';
        continue; # Fall through
    }
    when ($_ < 100) {
        say '$foo is numerically less than 100';
    }
    when (\&complicated_check) {
        say 'a complicated check for $foo is true';
    }
    default {
        die q(I don't know what to do with $foo);
    }
}
```

Before Perl 5.18, `given(EXPR)` assigned the value of *EXPR* to merely a lexically scoped ***copy*** (!) of `$_`, not a dynamically scoped alias the way `foreach` does. That made it similar to

```
do { my $_ = EXPR; ... }
```

except that the block was automatically broken out of by a successful `when` or an explicit `break`. Because it was only a copy, and because it was only lexically scoped, not dynamically scoped, you could not do the things with it that you are used to in a `foreach` loop. In particular, it did not work for arbitrary function calls if those functions might try to access `$_`. Best stick to `foreach` for that.

Most of the power comes from the implicit smartmatching that can sometimes apply. Most of the time, `when(EXPR)` is treated as an implicit smartmatch of `$_`, that is, `$_ ~~ EXPR`. (See "Smartmatch Operator" in perlop for more information on smartmatching.) But when *EXPR* is one of the 10 exceptional cases (or things like them) listed below, it is used directly as a boolean.

1. A user-defined subroutine call or a method invocation.

2. A regular expression match in the form of `/REGEX/`, `$foo =~ /REGEX/`, or `$foo =~ EXPR`. Also, a negated regular expression match in the form `!/REGEX/`, `$foo !~ /REGEX/`, or `$foo !~ EXPR`.

3. A smart match that uses an explicit `~~` operator, such as `EXPR ~~ EXPR`.

   NOTE: You will often have to use `$c ~~ $_` because the default case uses `$_ ~~ $c` , which is frequentlythe opposite of what you want.

4. A boolean comparison operator such as `$_ < 10` or `$x eq "abc"`. The relational operators that this applies to are the six numeric comparisons (`<`, `>`, `<=`, `>=`, `==`, and `!=`), and the six string comparisons (`lt`, `gt`, `le`, `ge`, `eq`, and `ne`).

5. At least the three builtin functions `defined(...)`, `exists(...)`, and `eof(...)`. We might someday add more of these later if we think of them.

6. A negated expression, whether `!(EXPR)` or `not(EXPR)`, or a logical exclusive-or, `(EXPR1) xor (EXPR2)`. The bitwise versions (`~` and `^`) are not included.

7. A filetest operator, with exactly 4 exceptions: `-s`, `-M`, `-A`, and `-C`, as these return numerical values, not boolean ones. The `-z` filetest operator is not included in the exception list.

8. The `..` and `...` flip-flop operators. Note that the `...` flip-flop operator is completely different from the `...` elliptical statement just described.

In those 8 cases above, the value of EXPR is used directly as a boolean, so no smartmatching is done. You may think of `when` as a smartsmartmatch.

Furthermore, Perl inspects the operands of logical operators to decide whether to use smartmatching for each one by applying the above test to the operands:

9. If EXPR is `EXPR1 && EXPR2` or `EXPR1 and EXPR2`, the test is applied *recursively* to both EXPR1 and EXPR2. Only if *both* operands also pass the test, *recursively*, will the expression be treated as boolean. Otherwise, smartmatching is used.

10. If EXPR is `EXPR1 || EXPR2`, `EXPR1 // EXPR2`, or `EXPR1 or EXPR2`, the test is applied *recursively* to EXPR1 only (which might itself be a higher-precedence AND operator, for example, and thus subject to the previous rule), not to EXPR2. If EXPR1 is to use smartmatching, then EXPR2 also does so, no matter what EXPR2 contains. But if EXPR2 does not get to use smartmatching, then the second argument will not be either. This is quite different from the `&&` case just described, so be careful.

These rules are complicated, but the goal is for them to do what you want (even if you don't quite understand why they are doing it). For example:

```
when (/^\d+$/ && $_ < 75) { ... }
```

will be treated as a boolean match because the rules say both a regex match and an explicit test on `$_` will be treated as boolean.

Also:

```
when ([qw(foo bar)] && /baz/) { ... }
```

will use smartmatching because only *one* of the operands is a boolean: the other uses smartmatching, and that wins.

Further:

```
when ([qw(foo bar)] || /^baz/) { ... }
```

will use smart matching (only the first operand is considered), whereas

```
when (/^baz/ || [qw(foo bar)]) { ... }
```

will test only the regex, which causes both operands to be treated as boolean. Watch out for this one, then, because an arrayref is always a true value, which makes it effectively redundant. Not a good idea.

Tautologous boolean operators are still going to be optimized away. Don't be tempted to write

```
when ("foo" or "bar") { ... }
```

This will optimize down to `"foo"`, so `"bar"` will never be considered (even though the rules say to use a smartmatch on `"foo"`). For an alternation like this, an array ref will work, because this will instigate

smartmatching:

```
when ([qw(foo bar)] { ... }
```

This is somewhat equivalent to the C−style switch statement's fallthrough functionality (not to be confused with *Perl's* fallthrough functionality — see below), wherein the same block is used for several `case` statements.

Another useful shortcut is that, if you use a literal array or hash as the argument to `given`, it is turned into a reference. So `given(@foo)` is the same as `given(\@foo)`, for example.

`default` behaves exactly like `when(1 == 1)`, which is to say that it always matches.

Breaking out

You can use the `break` keyword to break out of the enclosing `given` block. Every `when` block is implicitly ended with a `break`.

Fall-through

You can use the `continue` keyword to fall through from one case to the next immediate `when` or `default`:

```
given($foo) {
    when (/x/) { say '$foo contains an x'; continue }
    when (/y/) { say '$foo contains a y'            }
    default    { say '$foo does not contain a y'    }
}
```

Return value

When a `given` statement is also a valid expression (for example, when it's the last statement of a block), it evaluates to:

- An empty list as soon as an explicit `break` is encountered.
- The value of the last evaluated expression of the successful `when`/`default` clause, if there happens to be one.
- The value of the last evaluated expression of the `given` block if no condition is true.

In both last cases, the last expression is evaluated in the context that was applied to the `given` block.

Note that, unlike `if` and `unless`, failed `when` statements always evaluate to an empty list.

```
my $price = do {
    given ($item) {
        when (["pear", "apple"]) { 1 }
        break when "vote";      # My vote cannot be bought
        1e10  when /Mona Lisa/;
        "unknown";
    }
};
```

Currently, `given` blocks can't always be used as proper expressions. This may be addressed in a future version of Perl.

Switching in a loop

Instead of using `given()`, you can use a `foreach()` loop. For example, here's one way to count how many times a particular string occurs in an array:

```
use v5.10.1;
my $count = 0;
for (@array) {
    when ("foo") { ++$count }
}
print "\@array contains $count copies of 'foo'\n";
```

Or in a more recent version:

```
use v5.14;
my $count = 0;
for (@array) {
    ++$count when "foo";
}
print "\@array contains $count copies of 'foo'\n";
```

At the end of all `when` blocks, there is an implicit `next`. You can override that with an explicit `last` if you're interested in only the first match alone.

This doesn't work if you explicitly specify a loop variable, as in `for $item (@array)`. You have to use the default variable `$_`.

Differences from Perl 6

The Perl 5 smartmatch and `given`/`when` constructs are not compatible with their Perl 6 analogues. The most visible difference and least important difference is that, in Perl 5, parentheses are required around the argument to `given()` and `when()` (except when this last one is used as a statement modifier). Parentheses in Perl 6 are always optional in a control construct such as `if()`, `while()`, or `when()`; they can't be made optional in Perl 5 without a great deal of potential confusion, because Perl 5 would parse the expression

```
given $foo {
    ...
}
```

as though the argument to `given` were an element of the hash `%foo`, interpreting the braces as hash-element syntax.

However, their are many, many other differences. For example, this works in Perl 5:

```
use v5.12;
my @primary = ("red", "blue", "green");

if (@primary ~~ "red") {
    say "primary smartmatches red";
}

if ("red" ~~ @primary) {
    say "red smartmatches primary";
}

say "that's all, folks!";
```

But it doesn't work at all in Perl 6. Instead, you should use the (parallelizable) `any` operator:

```
if any(@primary) eq "red" {
    say "primary smartmatches red";
}

if "red" eq any(@primary) {
    say "red smartmatches primary";
}
```

The table of smartmatches in "Smartmatch Operator" in perlop is not identical to that proposed by the Perl 6 specification, mainly due to differences between Perl 6's and Perl 5's data models, but also because the Perl 6 spec has changed since Perl 5 rushed into early adoption.

In Perl 6, `when()` will always do an implicit smartmatch with its argument, while in Perl 5 it is convenient (albeit potentially confusing) to suppress this implicit smartmatch in various rather loosely-defined situations, as roughly outlined above. (The difference is largely because Perl 5 does not have, even internally, a boolean type.)

NAME

perldata – Perl data types

DESCRIPTION

Variable names

Perl has three built-in data types: scalars, arrays of scalars, and associative arrays of scalars, known as "hashes". A scalar is a single string (of any size, limited only by the available memory), number, or a reference to something (which will be discussed in perlref). Normal arrays are ordered lists of scalars indexed by number, starting with 0. Hashes are unordered collections of scalar values indexed by their associated string key.

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

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

Scalar values are always named with '$', even when referring to a scalar that is part of an array or a hash. The '$' symbol works semantically like the English word "the" in that it indicates a single value is expected.

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

Entire arrays (and slices of arrays and hashes) are denoted by '@', which works much as the word "these" or "those" does in English, in that it indicates multiple values are expected.

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

Entire hashes are denoted by '%':

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

In addition, subroutines are named with an initial '&', though this is optional when unambiguous, just as the word "do" is often redundant in English. Symbol table entries can be named with an initial '*', but you don't really care about that yet (if ever :–).

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

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

It is possible to replace such an alphanumeric name with an expression that returns a reference to the

appropriate type. For a description of this, see perlref.

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

Identifier parsing

Up until Perl 5.18, the actual rules of what a valid identifier was were a bit fuzzy. However, in general, anything defined here should work on previous versions of Perl, while the opposite — edge cases that work in previous versions, but aren't defined here — probably won't work on newer versions. As an important side note, please note that the following only applies to bareword identifiers as found in Perl source code, not identifiers introduced through symbolic references, which have much fewer restrictions. If working under the effect of the `use utf8;` pragma, the following rules apply:

```
/ (?[ ( \p{Word} & \p{XID_Start} ) + [_] ])
  (?[ ( \p{Word} & \p{XID_Continue} ) ]) *    /x
```

That is, a "start" character followed by any number of "continue" characters. Perl requires every character in an identifier to also match `\w` (this prevents some problematic cases); and Perl additionally accepts identfier names beginning with an underscore.

If not under `use utf8`, the source is treated as ASCII + 128 extra generic characters, and identifiers should match

```
/ (?aa) (?!\d) \w+ /x
```

That is, any word character in the ASCII range, as long as the first character is not a digit.

There are two package separators in Perl: A double colon (`::`) and a single quote (`'`). Normal identifiers can start or end with a double colon, and can contain several parts delimited by double colons. Single quotes have similar rules, but with the exception that they are not legal at the end of an identifier: That is, `$'foo` and `$foo'bar` are legal, but `$foo'bar'` is not.

Additionally, if the identifier is preceded by a sigil — that is, if the identifier is part of a variable name — it may optionally be enclosed in braces.

While you can mix double colons with singles quotes, the quotes must come after the colons: `$::::'foo` and `$foo::'bar` are legal, but `$::'::foo` and `$foo'::bar` are not.

Put together, a grammar to match a basic identifier becomes

```
/
  (?(DEFINE)
      (?<variable>
          (?&sigil)
          (?:
                  (?&normal_identifier)
              |   \{ \s* (?&normal_identifier) \s* \}
          )
      )
      (?<normal_identifier>
          (?: :: )* '?
           (?&basic_identifier)
           (?: (?= (?: :: )+ '? | (?: :: )* ' ) (?&normal_identifier) )?
          (?: :: )*
      )
      (?<basic_identifier>
        # is use utf8 on?
          (?(?{ (caller(0))[8] & $utf8::hint_bits })
              (?&Perl_XIDS) (?&Perl_XIDC)*
            | (?aa) (?!\d) \w+
          )
      )
      (?<sigil> [&*\$\@\%])
      (?<Perl_XIDS> (?[ ( \p{Word} & \p{XID_Start} ) + [_] ]) )
```

```
        (?<Perl_XIDC> (?[ \p{Word} & \p{XID_Continue} ]) )
    )
   /x
```

Meanwhile, special identifiers don't follow the above rules; For the most part, all of the identifiers in this category have a special meaning given by Perl. Because they have special parsing rules, these generally can't be fully-qualified. They come in six forms (but don't use forms 5 and 6):

1. A sigil, followed solely by digits matching `\p{POSIX_Digit}`, like `$0`, `$1`, or `$10000`.
2. A sigil followed by a single character matching the `\p{POSIX_Punct}` property, like `$!` or `%+`, except the character `"{"` doesn't work.
3. A sigil, followed by a caret and any one of the characters `[][A-Z^_?\]`, like `$^V` or `$^]`.
4. Similar to the above, a sigil, followed by bareword text in braces, where the first character is a caret. The next character is any one of the characters `[][A-Z^_?\]`, followed by ASCII word characters. An example is `${^GLOBAL_PHASE}`.
5. A sigil, followed by any single character in the range `[\xA1-\xAC\xAE-\xFF]` when not under `"use utf8"`. (Under `"use utf8"`, the normal identifier rules given earlier in this section apply.) Use of non-graphic characters (the C1 controls, the NO-BREAK SPACE, and the SOFT HYPHEN) has been disallowed since v5.26.0. The use of the other characters is unwise, as these are all reserved to have special meaning to Perl, and none of them currently do have special meaning, though this could change without notice.

   Note that an implication of this form is that there are identifiers only legal under `"use utf8"`, and vice-versa, for example the identifier `$état` is legal under `"use utf8"`, but is otherwise considered to be the single character variable `$é` followed by the bareword `"tat"`, the combination of which is a syntax error.
6. This is a combination of the previous two forms. It is valid only when not under `"use utf8"` (normal identifier rules apply when under `"use utf8"`). The form is a sigil, followed by text in braces, where the first character is any one of the characters in the range `[\x80-\xFF]` followed by ASCII word characters up to the trailing brace.

   The same caveats as the previous form apply: The non-graphic characters are no longer allowed with "use utf8", it is unwise to use this form at all, and utf8ness makes a big difference.

Prior to Perl v5.24, non-graphical ASCII control characters were also allowed in some situations; this had been deprecated since v5.20.

Context

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

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

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

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

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

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

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

When you use the `use warnings` pragma or Perl's **-w** command-line option, you may see warnings about useless uses of constants or functions in "void context". Void context just means the value has been discarded, such as a statement containing only `"fred";` or `getpwuid(0);`. It still counts as scalar context for functions that care whether or not they're being called in list context.

User-defined subroutines may choose to care whether they are being called in a void, scalar, or list context. Most subroutines do not need to bother, though. That's because both scalars and lists are automatically interpolated into lists. See "wantarray" in perlfunc for how you would dynamically discern your function's calling context.

Scalar values

All data in Perl is a scalar, an array of scalars, or a hash of scalars. A scalar may contain one single value in any of three different flavors: a number, a string, or a reference. In general, conversion from one form to another is transparent. Although a scalar may not directly hold multiple values, it may contain a reference to an array or hash which in turn contains multiple values.

Scalars aren't necessarily one thing or another. There's no place to declare a scalar variable to be of type "string", type "number", type "reference", or anything else. Because of the automatic conversion of scalars, operations that return scalars don't need to care (and in fact, cannot care) whether their caller is looking for a string, a number, or a reference. Perl is a contextually polymorphic language whose scalars can be strings, numbers, or references (which includes objects). Although strings and numbers are considered pretty much the same thing for nearly all purposes, references are strongly-typed, uncastable pointers with builtin reference-counting and destructor invocation.

A scalar value is interpreted as FALSE in the Boolean sense if it is undefined, the null string or the number 0 (or its string equivalent, "0"), and TRUE if it is anything else. The Boolean context is just a special kind of scalar context where no conversion to a string or a number is ever performed.

There are actually two varieties of null strings (sometimes referred to as "empty" strings), a defined one and an undefined one. The defined version is just a string of length zero, such as `""`. The undefined version is the value that indicates that there is no real value for something, such as when there was an error, or at end of file, or when you refer to an uninitialized variable or element of an array or hash. Although in early versions of Perl, an undefined scalar could become defined when first used in a place expecting a defined value, this no longer happens except for rare cases of autovivification as explained in perlref. You can use the *defined()* operator to determine whether a scalar value is defined (this has no meaning on arrays or hashes), and the *undef()* operator to produce an undefined value.

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

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

That method may be best because otherwise you won't treat IEEE notations like `NaN` or `Infinity` properly. At other times, you might prefer to determine whether string data can be used numerically by calling the *POSIX::strtod()* function or by inspecting your string with a regular expression (as documented in perlre).

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

The length of an array is a scalar value. You may find the length of array `@days` by evaluating `$#days`, as in **csh**. However, this isn't the length of the array; it's the subscript of the last element, which is a different value since there is ordinarily a 0th element. Assigning to `$#days` actually changes the length of the array. Shortening an array this way destroys intervening values. Lengthening an array that was previously shortened does not recover values that were in those elements.

You can also gain some minuscule measure of efficiency by pre-extending an array that is going to get big. You can also extend an array by assigning to an element that is off the end of the array. You can truncate an array down to nothing by assigning the null list () to it. The following are equivalent:

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

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

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

Some programmers choose to use an explicit conversion so as to leave nothing to doubt:

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

If you evaluate a hash in scalar context, it returns false if the hash is empty. If there are any key/value pairs, it returns true. A more precise definition is version dependent.

Prior to Perl 5.25 the value returned was a string consisting of the number of used buckets and the number of allocated buckets, separated by a slash. This is pretty much useful only to find out whether Perl's internal hashing algorithm is performing poorly on your data set. For example, you stick 10,000 things in a hash, but evaluating `%HASH` in scalar context reveals `"1/16"`, which means only one out of sixteen buckets has been touched, and presumably contains all 10,000 of your items. This isn't supposed to happen.

As of Perl 5.25 the return was changed to be the count of keys in the hash. If you need access to the old behavior you can use `Hash::Util::bucket_ratio()` instead.

If a tied hash is evaluated in scalar context, the `SCALAR` method is called (with a fallback to `FIRSTKEY`).

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

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

Scalar value constructors

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

```
12345
12345.67
.23E-10             # a very small number
3.14_15_92          # a very important number
4_294_967_296       # underscore for legibility
0xff                # hex
0xdead_beef         # more hex
0377                # octal (only numbers, begins with 0)
0b011011            # binary
0x1.999ap-4         # hexadecimal floating point (the 'p' is required)
```

You are allowed to use underscores (underbars) in numeric literals between digits for legibility (but not multiple underscores in a row: `23__500` is not legal; `23_500` is). You could, for example, group binary digits by threes (as for a Unix-style mode argument such as 0b110_100_100) or by fours (to represent nibbles, as in 0b1010_0110) or in other groups.

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

Hexadecimal, octal, or binary, representations in string literals (e.g. ’0xff’) are not automatically converted to their integer representation. The *hex()* and *oct()* functions make these conversions for you. See ‘‘hex’’ in perlfunc and ‘‘oct’’ in perlfunc for more details.

Hexadecimal floating point can start just like a hexadecimal literal, and it can be followed by an optional fractional hexadecimal part, but it must be followed by `p`, an optional sign, and a power of two. The format is useful for accurately presenting floating point values, avoiding conversions to or from decimal floating

point, and therefore avoiding possible loss in precision. Notice that while most current platforms use the 64−bit IEEE 754 floating point, not all do. Another potential source of (low-order) differences are the floating point rounding modes, which can differ between CPUs, operating systems, and compilers, and which Perl doesn't control.

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

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

There is no double interpolation in Perl, so the `$100` is left as is.

By default floating point numbers substituted inside strings use the dot (".") as the decimal separator. If `use locale` is in effect, and *POSIX::setlocale()* has been called, the character used for the decimal separator is affected by the LC_NUMERIC locale. See perllocale and POSIX.

As in some shells, you can enclose the variable name in braces to disambiguate it from following alphanumerics (and underscores). You must also do this when interpolating a variable into a string to separate the variable name from a following double-colon or an apostrophe, since these would be otherwise treated as a package separator:

```
$who = "Larry";
print PASSWD "${who}::0:0:Superuser:/:/bin/perl\n";
print "We use ${who}speak when ${who}'s here.\n";
```

Without the braces, Perl would have looked for a `$whospeak`, a `$who::0`, and a `$who's` variable. The last two would be the `$0` and the `$s` variables in the (presumably) non-existent package `who`.

In fact, a simple identifier within such curlies is forced to be a string, and likewise within a hash subscript. Neither need quoting. Our earlier example, `$days{'Feb'}` can be written as `$days{Feb}` and the quotes will be assumed automatically. But anything more complicated in the subscript will be interpreted as an expression. This means for example that `$version{2.0}++` is equivalent to `$version{2}++`, not to `$version{'2.0'}++`.

Special floating point: infinity (Inf) and not-a-number (NaN)

Floating point values include the special values `Inf` and `NaN`, for infinity and not-a-number. The infinity can be also negative.

The infinity is the result of certain math operations that overflow the floating point range, like 9**9**9. The not-a-number is the result when the result is undefined or unrepresentable. Though note that you cannot get `NaN` from some common "undefined" or "out-of-range" operations like dividing by zero, or square root of a negative number, since Perl generates fatal errors for those.

The infinity and not-a-number have their own special arithmetic rules. The general rule is that they are "contagious": `Inf` plus one is `Inf`, and `NaN` plus one is `NaN`. Where things get interesting is when you combine infinities and not-a-numbers: `Inf` minus `Inf` and `Inf` divided by `Inf` are `NaN` (while `Inf` plus `Inf` is `Inf` and `Inf` times `Inf` is `Inf`). `NaN` is also curious in that it does not equal any number, *including* itself: `NaN` != `NaN`.

Perl doesn't understand `Inf` and `NaN` as numeric literals, but you can have them as strings, and Perl will convert them as needed: "Inf" + 1. (You can, however, import them from the POSIX extension; `use POSIX qw(Inf NaN);` and then use them as literals.)

Note that on input (string to number) Perl accepts `Inf` and `NaN` in many forms. Case is ignored, and the Win32−specific forms like `1.#INF` are understood, but on output the values are normalized to `Inf` and `NaN`.

Version Strings

A literal of the form `v1.20.300.4000` is parsed as a string composed of characters with the specified ordinals. This form, known as v−strings, provides an alternative, more readable way to construct strings, rather than use the somewhat less readable interpolation form `"\x{1}\x{14}\x{12c}\x{fa0}"`.

This is useful for representing Unicode strings, and for comparing version "numbers" using the string comparison operators, `cmp`, `gt`, `lt` etc. If there are two or more dots in the literal, the leading `v` may be omitted.

```
print v9786;              # prints SMILEY, "\x{263a}"
print v102.111.111;       # prints "foo"
print 102.111.111;        # same
```

Such literals are accepted by both `require` and `use` for doing a version check. Note that using the v-strings for IPv4 addresses is not portable unless you also use the *inet_aton()/inet_ntoa()* routines of the Socket package.

Note that since Perl 5.8.1 the single-number v-strings (like `v65`) are not v-strings before the => operator (which is usually used to separate a hash key from a hash value); instead they are interpreted as literal strings ('v65'). They were v-strings from Perl 5.6.0 to Perl 5.8.0, but that caused more confusion and breakage than good. Multi-number v-strings like `v65.66` and `65.66.67` continue to be v-strings always.

Special Literals

The special literals __FILE__, __LINE__, and __PACKAGE__ represent the current filename, line number, and package name at that point in your program. __SUB__ gives a reference to the current subroutine. They may be used only as separate tokens; they will not be interpolated into strings. If there is no current package (due to an empty `package;` directive), __PACKAGE__ is the undefined value. (But the empty `package;` is no longer supported, as of version 5.10.) Outside of a subroutine, __SUB__ is the undefined value. __SUB__ is only available in 5.16 or higher, and only with a `use v5.16` or `use feature "current_sub"` declaration.

The two control characters ^D and ^Z, and the tokens __END__ and __DATA__ may be used to indicate the logical end of the script before the actual end of file. Any following text is ignored.

Text after __DATA__ may be read via the filehandle `PACKNAME::DATA`, where `PACKNAME` is the package that was current when the __DATA__ token was encountered. The filehandle is left open pointing to the line after __DATA__. The program should `close DATA` when it is done reading from it. (Leaving it open leaks filehandles if the module is reloaded for any reason, so it's a safer practice to close it.) For compatibility with older scripts written before __DATA__ was introduced, __END__ behaves like __DATA__ in the top level script (but not in files loaded with `require` or `do`) and leaves the remaining contents of the file accessible via `main::DATA`.

See SelfLoader for more description of __DATA__, and an example of its use. Note that you cannot read from the DATA filehandle in a BEGIN block: the BEGIN block is executed as soon as it is seen (during compilation), at which point the corresponding __DATA__ (or __END__) token has not yet been seen.

Barewords

A word that has no other interpretation in the grammar will be treated as if it were a quoted string. These are known as "barewords". As with filehandles and labels, a bareword that consists entirely of lowercase letters risks conflict with future reserved words, and if you use the `use warnings` pragma or the **-w** switch, Perl will warn you about any such words. Perl limits barewords (like identifiers) to about 250 characters. Future versions of Perl are likely to eliminate these arbitrary limitations.

Some people may wish to outlaw barewords entirely. If you say

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

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

Array Interpolation

Arrays and slices are interpolated into double-quoted strings by joining the elements with the delimiter specified in the `$"` variable (`$LIST_SEPARATOR` if "use English;" is specified), space by default. The following are equivalent:

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

```
system "echo @ARGV";
```

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

If you're looking for the information on how to use here-documents, which used to be here, that's been moved to "Quote and Quote-like Operators" in perlop.

List value constructors

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

```
(LIST)
```

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

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

assigns the entire list value to array `@foo`, but

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

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

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

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

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

To use a here-document to assign an array, one line per element, you might use an approach like this:

```
@sauces = <<End_Lines =~ m/(\S.*\S)/g;
    normal tomato
    spicy tomato
    green chile
    pesto
    white wine
End_Lines
```

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

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

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

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

This interpolation combines with the facts that the opening and closing parentheses are optional (except when necessary for precedence) and lists may end with an optional comma to mean that multiple commas within lists are legal syntax. The list `1,,3` is a concatenation of two lists, `1,` and `3`, the first of which ends with that optional comma. `1,,3` is `(1,),(3)` is `1,3` (And similarly for `1,,,3` is `(1,),(,),3` is `1,3` and so on.) Not that we'd advise you to use this obfuscation.

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

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

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

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

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

Lists may be assigned to only when each element of the list is itself legal to assign to:

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

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

An exception to this is that you may assign to `undef` in a list. This is useful for throwing away some of the return values of a function:

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

As of Perl 5.22, you can also use `(undef)x2` instead of `undef, undef`. (You can also do `($x) x 2`, which is less useful, because it assigns to the same variable twice, clobbering the first value assigned.)

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

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

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

It's also the source of a useful idiom for executing a function or performing an operation in list context and then counting the number of return values, by assigning to an empty list and then using that assignment in scalar context. For example, this code:

```
$count = () = $string =~ /\d+/g;
```

will place into `$count` the number of digit groups found in `$string`. This happens because the pattern match is in list context (since it is being assigned to the empty list), and will therefore return a list of all matching parts of the string. The list assignment in scalar context will translate that into the number of elements (here, the number of times the pattern matched) and assign that to `$count`. Note that simply using

```
$count = $string =~ /\d+/g;
```

would not have worked, since a pattern match in scalar context will only return true or false, rather than a count of matches.

The final element of a list assignment may be an array or a hash:

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

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

A hash can be initialized using a literal list holding pairs of items to be interpreted as a key and a value:

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

While literal lists and named arrays are often interchangeable, that's not the case for hashes. Just because you can subscript a list value like a normal array does not mean that you can subscript a list value as a hash. Likewise, hashes included as parts of other lists (including parameters lists and return lists from functions)

always flatten out into key/value pairs. That's why it's good to use references sometimes.

It is often more readable to use the => operator between key/value pairs. The => operator is mostly just a more visually distinctive synonym for a comma, but it also arranges for its left-hand operand to be interpreted as a string if it's a bareword that would be a legal simple identifier. => doesn't quote compound identifiers, that contain double colons. This makes it nice for initializing hashes:

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

or for initializing hash references to be used as records:

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

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

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

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

If a key appears more than once in the initializer list of a hash, the last occurrence wins:

```
%circle = (
              center => [5, 10],
              center => [27, 9],
              radius => 100,
              color => [0xDF, 0xFF, 0x00],
              radius => 54,
);

# same as
%circle = (
              center => [27, 9],
              color => [0xDF, 0xFF, 0x00],
              radius => 54,
);
```

This can be used to provide overridable configuration defaults:

```
# values in %args take priority over %config_defaults
%config = (%config_defaults, %args);
```

Subscripts

An array can be accessed one scalar at a time by specifying a dollar sign (`$`), then the name of the array (without the leading `@`), then the subscript inside square brackets. For example:

```
@myarray = (5, 50, 500, 5000);
print "The Third Element is", $myarray[2], "\n";
```

The array indices start with 0. A negative subscript retrieves its value from the end. In our example, `$myarray[-1]` would have been 5000, and `$myarray[-2]` would have been 500.

Hash subscripts are similar, only instead of square brackets curly brackets are used. For example:

```
%scientists =
(
    "Newton" => "Isaac",
    "Einstein" => "Albert",
    "Darwin" => "Charles",
    "Feynman" => "Richard",
);

print "Darwin's First Name is ", $scientists{"Darwin"}, "\n";
```

You can also subscript a list to get a single element from it:

```
$dir = (getpwnam("daemon"))[7];
```

Multi-dimensional array emulation

Multidimensional arrays may be emulated by subscripting a hash with a list. The elements of the list are joined with the subscript separator (see "$;" in perlvar).

```
$foo{$a,$b,$c}
```

is equivalent to

```
$foo{join($;, $a, $b, $c)}
```

The default subscript separator is "\034", the same as SUBSEP in **awk**.

Slices

A slice accesses several elements of a list, an array, or a hash simultaneously using a list of subscripts. It's more convenient than writing out the individual elements as a list of separate scalar values.

```
($him, $her)   = @folks[0,-1];              # array slice
@them          = @folks[0 .. 3];            # array slice
($who, $home)  = @ENV{"USER", "HOME"};      # hash slice
($uid, $dir)   = (getpwnam("daemon"))[2,7]; # list slice
```

Since you can assign to a list of variables, you can also assign to an array or hash slice.

```
@days[3..5]    = qw/Wed Thu Fri/;
@colors{'red','blue','green'}
               = (0xff0000, 0x0000ff, 0x00ff00);
@folks[0, -1]  = @folks[-1, 0];
```

The previous assignments are exactly equivalent to

```
($days[3], $days[4], $days[5]) = qw/Wed Thu Fri/;
($colors{'red'}, $colors{'blue'}, $colors{'green'})
               = (0xff0000, 0x0000ff, 0x00ff00);
($folks[0], $folks[-1]) = ($folks[-1], $folks[0]);
```

Since changing a slice changes the original array or hash that it's slicing, a `foreach` construct will alter some — or even all — of the values of the array or hash.

```
foreach (@array[ 4 .. 10 ]) { s/peter/paul/ }

foreach (@hash{qw[key1 key2]}) {
    s/^\s+//;           # trim leading whitespace
    s/\s+$//;           # trim trailing whitespace
    s/(\w+)/\u\L$1/g;   # "titlecase" words
}
```

As a special exception, when you slice a list (but not an array or a hash), if the list evaluates to empty, then taking a slice of that empty list will always yield the empty list in turn. Thus:

```
@a = ()[0,1];           # @a has no elements
@b = (@a)[0,1];         # @b has no elements
@c = (sub{}->())[0,1]; # @c has no elements
@d = ('a','b')[0,1];    # @d has two elements
@e = (@d)[0,1,8,9];     # @e has four elements
@f = (@d)[8,9];         # @f has two elements
```

This makes it easy to write loops that terminate when a null list is returned:

```
while ( ($home, $user) = (getpwent)[7,0] ) {
    printf "%-8s %s\n", $user, $home;
}
```

As noted earlier in this document, the scalar sense of list assignment is the number of elements on the right-hand side of the assignment. The null list contains no elements, so when the password file is exhausted, the result is 0, not 2.

Slices in scalar context return the last item of the slice.

```
@a = qw/first second third/;
%h = (first => 'A', second => 'B');
$t = @a[0, 1];                  # $t is now 'second'
$u = @h{'first', 'second'};     # $u is now 'B'
```

If you're confused about why you use an '@' there on a hash slice instead of a '%', think of it like this. The type of bracket (square or curly) governs whether it's an array or a hash being looked at. On the other hand, the leading symbol ('$' or '@') on the array or hash indicates whether you are getting back a singular value (a scalar) or a plural one (a list).

Key/Value Hash Slices

Starting in Perl 5.20, a hash slice operation with the % symbol is a variant of slice operation returning a list of key/value pairs rather than just values:

```
%h = (blonk => 2, foo => 3, squink => 5, bar => 8);
%subset = %h{'foo', 'bar'}; # key/value hash slice
# %subset is now (foo => 3, bar => 8)
```

However, the result of such a slice cannot be localized, deleted or used in assignment. These are otherwise very much consistent with hash slices using the @ symbol.

Index/Value Array Slices

Similar to key/value hash slices (and also introduced in Perl 5.20), the % array slice syntax returns a list of index/value pairs:

```
@a = "a".."z";
@list = %a[3,4,6];
# @list is now (3, "d", 4, "e", 6, "g")
```

Typeglobs and Filehandles

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

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

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

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

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

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

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

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

or perhaps as a real reference, like this:

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

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

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

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

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

All functions that are capable of creating filehandles (*open()*, *opendir()*, *pipe()*, *socketpair()*, *sysopen()*, *socket()*, and *accept()*) automatically create an anonymous filehandle if the handle passed to them is an uninitialized scalar variable. This allows the constructs such as `open(my $fh, ...)` and `open(local $fh,...)` to be used to create filehandles that will conveniently be closed automatically when the scope ends, provided there are no other references to them. This largely eliminates the need for typeglobs when opening filehandles that must be passed around, as in the following example:

```
sub myopen {
    open my $fh, "@_"
         or die "Can't open '@_': $!";
    return $fh;
}

{
    my $f = myopen("</etc/motd");
    print <$f>;
    # $f implicitly closed here
}
```

Note that if an initialized scalar variable is used instead the result is different: `my $fh='zzz'; open($fh, ...)` is equivalent to `open( *{'zzz'}, ...)`. `use strict 'refs'` forbids such practice.

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

SEE ALSO

See perlvar for a description of Perl's built-in variables and a discussion of legal variable names. See perlref, perlsub, and "Symbol Tables" in perlmod for more discussion on typeglobs and the `*foo{THING}` syntax.

NAME

perlop – Perl operators and precedence

DESCRIPTION

In Perl, the operator determines what operation is performed, independent of the type of the operands. For example `$x + $y` is always a numeric addition, and if `$x` or `$y` do not contain numbers, an attempt is made to convert them to numbers first.

This is in contrast to many other dynamic languages, where the operation is determined by the type of the first argument. It also means that Perl has two versions of some operators, one for numeric and one for string comparison. For example `$x == $y` compares two numbers for equality, and `$x eq $y` compares two strings.

There are a few exceptions though: `x` can be either string repetition or list repetition, depending on the type of the left operand, and `&`, `|`, `^` and `~` can be either string or numeric bit operations.

Operator Precedence and Associativity

Operator precedence and associativity work in Perl more or less like they do in mathematics.

Operator precedence means some operators are evaluated before others. For example, in `2 + 4 * 5`, the multiplication has higher precedence so `4 * 5` is evaluated first yielding `2 + 20 == 22` and not `6 * 5 == 30`.

Operator associativity defines what happens if a sequence of the same operators is used one after another: whether the evaluator will evaluate the left operations first, or the right first. For example, in `8 - 4 - 2`, subtraction is left associative so Perl evaluates the expression left to right. `8 - 4` is evaluated first making the expression `4 - 2 == 2` and not `8 - 2 == 6`.

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

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

In the following sections, these operators are covered in detail, in the same order in which they appear in the table above.

Many operators can be overloaded for objects. See overload.

Terms and List Operators (Leftward)

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

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

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

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

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

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

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

Also note that

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

probably doesn't do what you expect at first glance. The parentheses enclose the argument list for `print` which is evaluated (printing the result of `$foo & 255`). Then one is added to the return value of `print` (usually 1). The result is something like this:

```
1 + 1, "\n";    # Obviously not what you meant.
```

To do what you meant properly, you must write:

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

See ''Named Unary Operators'' for more discussion of this.

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

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

The Arrow Operator

"`->`" is an infix dereference operator, just as it is in C and C++. If the right side is either a `[...]`, `{...}`, or a `(...)` subscript, then the left side must be either a hard or symbolic reference to an array, a hash, or a subroutine respectively. (Or technically speaking, a location capable of holding a hard reference, if it's an array or hash reference being used for assignment.) See perlreftut and perlref.

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

The dereferencing cases (as opposed to method-calling cases) are somewhat extended by the `postderef` feature. For the details of that feature, consult ''Postfix Dereference Syntax'' in perlref.

Auto-increment and Auto-decrement

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

```
$i = 0;  $j = 0;
print $i++;  # prints 0
print ++$j;  # prints 1
```

Note that just as in C, Perl doesn't define **when** the variable is incremented or decremented. You just know it will be done sometime before or after the value is returned. This also means that modifying a variable twice in the same statement will lead to undefined behavior. Avoid statements like:

```
$i = $i ++;
print ++ $i + $i ++;
```

Perl will not guarantee what the result of the above statements is.

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

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

`undef` is always treated as numeric, and in particular is changed to `0` before incrementing (so that a post-increment of an undef value will return `0` rather than `undef`).

The auto-decrement operator is not magical.

Exponentiation

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

Note that certain exponentiation expressions are ill-defined: these include `0**0`, `1**Inf`, and `Inf**0`. Do not expect any particular results from these special cases, the results are platform-dependent.

Symbolic Unary Operators

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

Unary `"-"` performs arithmetic negation if the operand is numeric, including any string that looks like a number. If the operand is an identifier, a string consisting of a minus sign concatenated with the identifier is returned. Otherwise, if the string starts with a plus or minus, a string starting with the opposite sign is returned. One effect of these rules is that `-bareword` is equivalent to the string `"-bareword"`. If, however, the string begins with a non-alphabetic character (excluding `"+"` or `"-"`), Perl will attempt to convert the string to a numeric, and the arithmetic negation is performed. If the string cannot be cleanly converted to a numeric, Perl will give the warning **Argument "the string" isn't numeric in negation (-) at ...**.

Unary `"~"` performs bitwise negation, that is, 1's complement. For example, `0666 & ~027` is 0640. (See also "Integer Arithmetic" and "Bitwise String Operators".) Note that the width of the result is platform-dependent: `~0` is 32 bits wide on a 32-bit platform, but 64 bits wide on a 64-bit platform, so if you are expecting a certain bit width, remember to use the `"&"` operator to mask off the excess bits.

When complementing strings, if all characters have ordinal values under 256, then their complements will, also. But if they do not, all characters will be in either 32- or 64-bit complements, depending on your architecture. So for example, `~"\x{3B1}"` is `"\x{FFFF_FC4E}"` on 32-bit machines and `"\x{FFFF_FFFF_FFFF_FC4E}"` on 64-bit machines.

If the experimental "bitwise" feature is enabled via `use feature 'bitwise'`, then unary `"~"` always treats its argument as a number, and an alternate form of the operator, `"~."`, always treats its argument as a string. So `~0` and `~"0"` will both give 2**32-1 on 32-bit platforms, whereas `~.0` and `~."0"` will both yield `"\xff"`. This feature produces a warning unless you use `no warnings 'experimental::bitwise'`.

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

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

Binding Operators

Binary "=~" binds a scalar expression to a pattern match. Certain operations search or modify the string $_ by default. This operator makes that kind of operation work on some other string. The right argument is a search pattern, substitution, or transliteration. The left argument is what is supposed to be searched, substituted, or transliterated instead of the default $_. When used in scalar context, the return value generally indicates the success of the operation. The exceptions are substitution (s///) and transliteration (y///) with the /r (non-destructive) option, which cause the **r**eturn value to be the result of the substitution. Behavior in list context depends on the particular operator. See "Regexp Quote-Like Operators" for details and perlretut for examples using these operators.

If the right argument is an expression rather than a search pattern, substitution, or transliteration, it is interpreted as a search pattern at run time. Note that this means that its contents will be interpolated twice, so

```
'\\' =~ q'\\';
```

is not ok, as the regex engine will end up trying to compile the pattern \, which it will consider a syntax error.

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

Binary "!~" with a non-destructive substitution (s///r) or transliteration (y///r) is a syntax error.

Multiplicative Operators

Binary "*" multiplies two numbers.

Binary "/" divides two numbers.

Binary "%" is the modulo operator, which computes the division remainder of its first argument with respect to its second argument. Given integer operands $m and $n: If $n is positive, then $m % $n is $m minus the largest multiple of $n less than or equal to $m. If $n is negative, then $m % $n is $m minus the smallest multiple of $n that is not less than $m (that is, the result will be less than or equal to zero). If the operands $m and $n are floating point values and the absolute value of $n (that is abs($n)) is less than (UV_MAX + 1), only the integer portion of $m and $n will be used in the operation (Note: here UV_MAX means the maximum of the unsigned integer type). If the absolute value of the right operand (abs($n)) is greater than or equal to (UV_MAX + 1), "%" computes the floating-point remainder $r in the equation ($r = $m - $i*$n) where $i is a certain integer that makes $r have the same sign as the right operand $n (**not** as the left operand $m like C function fmod()) and the absolute value less than that of $n. Note that when use integer is in scope, "%" gives you direct access to the modulo operator as implemented by your C compiler. This operator is not as well defined for negative operands, but it will execute faster.

Binary "x" is the repetition operator. In scalar context or if the left operand is not enclosed in parentheses, it returns a string consisting of the left operand repeated the number of times specified by the right operand. In list context, if the left operand is enclosed in parentheses or is a list formed by qw/*STRING*/, it repeats the list. If the right operand is zero or negative (raising a warning on negative), it returns an empty string or an empty list, depending on the context.

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

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

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

Additive Operators

Binary "+" returns the sum of two numbers.

Binary "-" returns the difference of two numbers.

Binary "." concatenates two strings.

Shift Operators

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

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

If `use integer` (see ‘‘Integer Arithmetic’’) is in force then signed C integers are used (*arithmetic shift*), otherwise unsigned C integers are used (*logical shift*), even for negative shiftees. In arithmetic right shift the sign bit is replicated on the left, in logical shift zero bits come in from the left.

Either way, the implementation isn't going to generate results larger than the size of the integer type Perl was built with (32 bits or 64 bits).

Shifting by negative number of bits means the reverse shift: left shift becomes right shift, right shift becomes left shift. This is unlike in C, where negative shift is undefined.

Shifting by more bits than the size of the integers means most of the time zero (all bits fall off), except that under `use integer` right overshifting a negative shiftee results in −1. This is unlike in C, where shifting by too many bits is undefined. A common C behavior is ‘‘shift by modulo wordbits’’, so that for example

```
1 >> 64 == 1 >> (64 % 64) == 1 >> 0 == 1  # Common C behavior.
```

but that is completely accidental.

If you get tired of being subject to your platform's native integers, the `use bigint` pragma neatly sidesteps the issue altogether:

```
print 20 << 20;  # 20971520
print 20 << 40;  # 5120 on 32-bit machines,
                 # 21990232555520 on 64-bit machines
use bigint;
print 20 << 100; # 25353012004564588029934064107520
```

Named Unary Operators

The various named unary operators are treated as functions with one argument, with optional parentheses.

If any list operator (`print()`, etc.) or any unary operator (`chdir()`, etc.) is followed by a left parenthesis as the next token, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call. For example, because named unary operators are higher precedence than `||`:

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

but, because `"*"` is higher precedence than named operators:

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

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

Regarding precedence, the filetest operators, like `-f`, `-M`, etc. are treated like named unary operators, but they don't follow this functional parenthesis rule. That means, for example, that `-f($file).".bak"` is equivalent to `-f "$file.bak"`.

See also ‘‘Terms and List Operators (Leftward)’’.

Relational Operators

Perl operators that return true or false generally return values that can be safely used as numbers. For example, the relational operators in this section and the equality operators in the next one return `1` for true and a special version of the defined empty string, `""`, which counts as a zero but is exempt from warnings

about improper numeric conversions, just as `"0 but true"` is.

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

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

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

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

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

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

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

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

Equality Operators

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

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

Binary `"<=>"` returns −1, 0, or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument. If your platform supports `NaN`'s (not-a-numbers) as numeric values, using them with `"<=>"` returns undef. `NaN` is not `"<"`, `"=="`, `">"`, `"<="` or `">="` anything (even `NaN`), so those 5 return false. `NaN != NaN` returns true, as does `NaN !=` *anything else*. If your platform doesn't support `NaN`'s then `NaN` is just a string with numeric value 0.

```
$ perl -le '$x = "NaN"; print "No NaN support here" if $x == $x'
$ perl -le '$x = "NaN"; print "NaN support here" if $x != $x'
```

(Note that the bigint, bigrat, and bignum pragmas all support `"NaN"`.)

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

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

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

Binary `"~~"` does a smartmatch between its arguments. Smart matching is described in the next section.

`"lt"`, `"le"`, `"ge"`, `"gt"` and `"cmp"` use the collation (sort) order specified by the current `LC_COLLATE` locale if a `use locale` form that includes collation is in effect. See perllocale. Do not mix these with Unicode, only use them with legacy 8−bit locale encodings. The standard `Unicode::Collate` and `Unicode::Collate::Locale` modules offer much more powerful solutions to collation issues.

For case-insensitive comparisions, look at the "fc" in perlfunc case-folding function, available in Perl v5.16 or later:

```
if ( fc($x) eq fc($y) ) { ... }
```

Smartmatch Operator

First available in Perl 5.10.1 (the 5.10.0 version behaved differently), binary `~~` does a "smartmatch" between its arguments. This is mostly used implicitly in the `when` construct described in perlsyn, although not all `when` clauses call the smartmatch operator. Unique among all of Perl's operators, the smartmatch operator can recurse. The smartmatch operator is experimental and its behavior is subject to change.

It is also unique in that all other Perl operators impose a context (usually string or numeric context) on their operands, autoconverting those operands to those imposed contexts. In contrast, smartmatch *infers* contexts from the actual types of its operands and uses that type information to select a suitable comparison mechanism.

The `~~` operator compares its operands "polymorphically", determining how to compare them according to their actual types (numeric, string, array, hash, etc.) Like the equality operators with which it shares the same precedence, `~~` returns 1 for true and `""` for false. It is often best read aloud as "in", "inside of", or "is contained in", because the left operand is often looked for *inside* the right operand. That makes the order of the operands to the smartmatch operand often opposite that of the regular match operator. In other words, the "smaller" thing is usually placed in the left operand and the larger one in the right.

The behavior of a smartmatch depends on what type of things its arguments are, as determined by the

following table. The first row of the table whose types apply determines the smartmatch behavior. Because what actually happens is mostly determined by the type of the second operand, the table is sorted on the right operand instead of on the left.

```
 Left      Right      Description and pseudocode
 ===============================================================
 Any       undef      check whether Any is undefined
                 like: !defined Any

 Any       Object     invoke ~~ overloading on Object, or die

 Right operand is an ARRAY:

 Left      Right      Description and pseudocode
 ===============================================================
 ARRAY1    ARRAY2     recurse on paired elements of ARRAY1 and ARRAY2[2]
                 like: (ARRAY1[0] ~~ ARRAY2[0])
                         && (ARRAY1[1] ~~ ARRAY2[1]) && ...
 HASH      ARRAY      any ARRAY elements exist as HASH keys
                 like: grep { exists HASH->{$_} } ARRAY
 Regexp    ARRAY      any ARRAY elements pattern match Regexp
                 like: grep { /Regexp/ } ARRAY
 undef     ARRAY      undef in ARRAY
                 like: grep { !defined } ARRAY
 Any       ARRAY      smartmatch each ARRAY element[3]
                 like: grep { Any ~~ $_ } ARRAY

 Right operand is a HASH:

 Left      Right      Description and pseudocode
 ===============================================================
 HASH1     HASH2      all same keys in both HASHes
                 like: keys HASH1 ==
                            grep { exists HASH2->{$_} } keys HASH1
 ARRAY     HASH       any ARRAY elements exist as HASH keys
                 like: grep { exists HASH->{$_} } ARRAY
 Regexp    HASH       any HASH keys pattern match Regexp
                 like: grep { /Regexp/ } keys HASH
 undef     HASH       always false (undef can't be a key)
                 like: 0 == 1
 Any       HASH       HASH key existence
                 like: exists HASH->{Any}

 Right operand is CODE:

 Left      Right      Description and pseudocode
 ===============================================================
 ARRAY     CODE       sub returns true on all ARRAY elements[1]
                 like: !grep { !CODE->($_) } ARRAY
 HASH      CODE       sub returns true on all HASH keys[1]
                 like: !grep { !CODE->($_) } keys HASH
 Any       CODE       sub passed Any returns true
                 like: CODE->(Any)
```

Right operand is a Regexp:

```
Left      Right      Description and pseudocode
===============================================================
ARRAY     Regexp     any ARRAY elements match Regexp
               like: grep { /Regexp/ } ARRAY
HASH      Regexp     any HASH keys match Regexp
               like: grep { /Regexp/ } keys HASH
Any       Regexp     pattern match
               like: Any =~ /Regexp/

Other:

Left      Right      Description and pseudocode
===============================================================
Object    Any        invoke ~~ overloading on Object,
                     or fall back to...

Any       Num        numeric equality
                like: Any == Num
Num       nummy[4]    numeric equality
                like: Num == nummy
undef     Any        check whether undefined
                like: !defined(Any)
Any       Any        string equality
                like: Any eq Any
```

Notes:

1. Empty hashes or arrays match.
2. That is, each element smartmatches the element of the same index in the other array.[3]
3. If a circular reference is found, fall back to referential equality.
4. Either an actual number, or a string that looks like one.

The smartmatch implicitly dereferences any non-blessed hash or array reference, so the *HASH* and *ARRAY* entries apply in those cases. For blessed references, the *Object* entries apply. Smartmatches involving hashes only consider hash keys, never hash values.

The "like" code entry is not always an exact rendition. For example, the smartmatch operator short-circuits whenever possible, but `grep` does not. Also, `grep` in scalar context returns the number of matches, but `~~` returns only true or false.

Unlike most operators, the smartmatch operator knows to treat `undef` specially:

```
use v5.10.1;
@array = (1, 2, 3, undef, 4, 5);
say "some elements undefined" if undef ~~ @array;
```

Each operand is considered in a modified scalar context, the modification being that array and hash variables are passed by reference to the operator, which implicitly dereferences them. Both elements of each pair are the same:

```
use v5.10.1;

my %hash = (red    => 1, blue   => 2, green  => 3,
            orange => 4, yellow => 5, purple => 6,
            black  => 7, grey   => 8, white  => 9);

my @array = qw(red blue green);

say "some array elements in hash keys" if  @array ~~  %hash;
say "some array elements in hash keys" if \@array ~~ \%hash;

say "red in array" if "red" ~~  @array;
say "red in array" if "red" ~~ \@array;
```

```
say "some keys end in e" if /e$/ ~~  %hash;
say "some keys end in e" if /e$/ ~~ \%hash;
```

Two arrays smartmatch if each element in the first array smartmatches (that is, is "in") the corresponding element in the second array, recursively.

```
use v5.10.1;
my @little = qw(red blue green);
my @bigger = ("red", "blue", [ "orange", "green" ] );
if (@little ~~ @bigger) {  # true!
    say "little is contained in bigger";
}
```

Because the smartmatch operator recurses on nested arrays, this will still report that "red" is in the array.

```
use v5.10.1;
my @array = qw(red blue green);
my $nested_array = [[[[[[[ @array ]]]]]]];
say "red in array" if "red" ~~ $nested_array;
```

If two arrays smartmatch each other, then they are deep copies of each others' values, as this example reports:

```
use v5.12.0;
my @a = (0, 1, 2, [3, [4, 5], 6], 7);
my @b = (0, 1, 2, [3, [4, 5], 6], 7);

if (@a ~~ @b && @b ~~ @a) {
    say "a and b are deep copies of each other";
}
elsif (@a ~~ @b) {
    say "a smartmatches in b";
}
elsif (@b ~~ @a) {
    say "b smartmatches in a";
}
else {
    say "a and b don't smartmatch each other at all";
}
```

If you were to set `$b[3] = 4`, then instead of reporting that "a and b are deep copies of each other", it now reports that `"b smartmatches in a"`. That's because the corresponding position in `@a` contains an array that (eventually) has a 4 in it.

Smartmatching one hash against another reports whether both contain the same keys, no more and no less. This could be used to see whether two records have the same field names, without caring what values those fields might have. For example:

```
use v5.10.1;
sub make_dogtag {
    state $REQUIRED_FIELDS = { name=>1, rank=>1, serial_num=>1 };

    my ($class, $init_fields) = @_;

    die "Must supply (only) name, rank, and serial number"
        unless $init_fields ~~ $REQUIRED_FIELDS;

    ...
}
```

However, this only does what you mean if `$init_fields` is indeed a hash reference. The condition `$init_fields ~~ $REQUIRED_FIELDS` also allows the strings `"name"`, `"rank"`, `"serial_num"` as well as any array reference that contains `"name"` or `"rank"` or `"serial_num"` anywhere to pass through.

The smartmatch operator is most often used as the implicit operator of a `when` clause. See the section on "Switch Statements" in perlsyn.

Smartmatching of Objects

To avoid relying on an object's underlying representation, if the smartmatch's right operand is an object that doesn't overload `~~`, it raises the exception "`Smartmatching a non-overloaded object breaks encapsulation`". That's because one has no business digging around to see whether something is "in" an object. These are all illegal on objects without a `~~` overload:

```
%hash ~~ $object
   42 ~~ $object
"fred" ~~ $object
```

However, you can change the way an object is smartmatched by overloading the `~~` operator. This is allowed to extend the usual smartmatch semantics. For objects that do have an `~~` overload, see overload.

Using an object as the left operand is allowed, although not very useful. Smartmatching rules take precedence over overloading, so even if the object in the left operand has smartmatch overloading, this will be ignored. A left operand that is a non-overloaded object falls back on a string or numeric comparison of whatever the `ref` operator returns. That means that

```
$object ~~ X
```

does *not* invoke the overload method with *X* as an argument. Instead the above table is consulted as normal, and based on the type of *X*, overloading may or may not be invoked. For simple strings or numbers, "in" becomes equivalent to this:

```
$object ~~ $number          ref($object) == $number
$object ~~ $string          ref($object) eq $string
```

For example, this reports that the handle smells IOish (but please don't really do this!):

```
use IO::Handle;
my $fh = IO::Handle->new();
if ($fh ~~ /\bIO\b/) {
    say "handle smells IOish";
}
```

That's because it treats `$fh` as a string like `"IO::Handle=GLOB(0x8039e0)"`, then pattern matches against that.

Bitwise And

Binary `"&"` returns its operands ANDed together bit by bit. Although no warning is currently raised, the result is not well defined when this operation is performed on operands that aren't either numbers (see "Integer Arithmetic") nor bitstrings (see "Bitwise String Operators").

Note that `"&"` has lower priority than relational operators, so for example the parentheses are essential in a test like

```
print "Even\n" if ($x & 1) == 0;
```

If the experimental "bitwise" feature is enabled via `use feature 'bitwise'`, then this operator always treats its operand as numbers. This feature produces a warning unless you also use `no warnings 'experimental::bitwise'`.

Bitwise Or and Exclusive Or

Binary `"|"` returns its operands ORed together bit by bit.

Binary `"^"` returns its operands XORed together bit by bit.

Although no warning is currently raised, the results are not well defined when these operations are performed on operands that aren't either numbers (see "Integer Arithmetic") nor bitstrings (see "Bitwise String Operators").

Note that `"|"` and `"^"` have lower priority than relational operators, so for example the parentheses are essential in a test like

```
print "false\n" if (8 | 2) != 10;
```

If the experimental "bitwise" feature is enabled via `use feature 'bitwise'`, then this operator

always treats its operand as numbers. This feature produces a warning unless you also use `no warnings 'experimental::bitwise'`.

C−style Logical And

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

C−style Logical Or

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

Logical Defined-Or

Although it has no direct equivalent in C, Perl's `//` operator is related to its C−style "or". In fact, it's exactly the same as `||`, except that it tests the left hand side's definedness instead of its truth. Thus, `EXPR1 // EXPR2` returns the value of `EXPR1` if it's defined, otherwise, the value of `EXPR2` is returned. (`EXPR1` is evaluated in scalar context, `EXPR2` in the context of `//` itself). Usually, this is the same result as `defined(EXPR1) ? EXPR1 : EXPR2` (except that the ternary-operator form can be used as a lvalue, while `EXPR1 // EXPR2` cannot). This is very useful for providing default values for variables. If you actually want to test if at least one of `$x` and `$y` is defined, use `defined($x // $y)`.

The `||`, `//` and `&&` operators return the last value evaluated (unlike C's `||` and `&&`, which return 0 or 1). Thus, a reasonably portable way to find out the home directory might be:

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

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

```
@a = @b || @c;            # This doesn't do the right thing
@a = scalar(@b) || @c;    # because it really means this.
@a = @b ? @b : @c;        # This works fine, though.
```

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

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

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

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

It would be even more readable to write that this way:

```
unless(unlink("alpha", "beta", "gamma")) {
    gripe();
    next LINE;
}
```

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

Range Operators

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

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

The range operator also works on strings, using the magical auto-increment, see below.

In scalar context, `".."` returns a boolean value. The operator is bistable, like a flip-flop, and emulates the line-range (comma) operator of **sed**, **awk**, and various editors. Each `".."` operator maintains its own boolean state, even across calls to a subroutine that contains it. It is false as long as its left operand is false. Once the left operand is true, the range operator stays true until the right operand is true, *AFTER* which the range operator becomes false again. It doesn't become false till the next time the range operator is evaluated. It can test the right operand and become false on the same evaluation it became true (as in **awk**), but it still returns true once. If you don't want it to test the right operand until the next evaluation, as in **sed**, just use three dots (`"..."`) instead of two. In all other regards, `"..."` behaves just like `".."` does.

The right operand is not evaluated while the operator is in the "false" state, and the left operand is not evaluated while the operator is in the "true" state. The precedence is a little lower than || and &&. The value returned is either the empty string for false, or a sequence number (beginning with 1) for true. The sequence number is reset for each range encountered. The final sequence number in a range has the string `"E0"` appended to it, which doesn't affect its numeric value, but gives you something to search for if you want to exclude the endpoint. You can exclude the beginning point by waiting for the sequence number to be greater than 1.

If either operand of scalar `".."` is a constant expression, that operand is considered true if it is equal (==) to the current input line number (the `$.` variable).

To be pedantic, the comparison is actually `int(EXPR) == int(EXPR)`, but that is only an issue if you use a floating point expression; when implicitly using `$.` as described in the previous paragraph, the comparison is `int(EXPR) == int($.)` which is only an issue when `$.` is set to a floating point value and you are not reading from a file. Furthermore, `"span" .. "spat"` or `2.18 .. 3.14` will not do what you want in scalar context because each of the operands are evaluated using their integer representation.

Examples:

As a scalar operator:

```
if (101 .. 200) { print; } # print 2nd hundred lines, short for
                           #  if ($. == 101 .. $. == 200) { print; }

next LINE if (1 .. /^$/);  # skip header lines, short for
                           #   next LINE if ($. == 1 .. /^$/);
                           # (typically in a loop labeled LINE)

s/^/> / if (/^$/ .. eof());  # quote body

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

Here's a simple example to illustrate the difference between the two range operators:

```
@lines = ("   - Foo",
          "01 - Bar",
          "1  - Baz",
          "   - Quux");

foreach (@lines) {
    if (/0/ .. /1/) {
        print "$_\n";
```

```
    }
}
```

This program will print only the line containing "Bar". If the range operator is changed to `...`, it will also print the "Baz" line.

And now some examples as a list operator:

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

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

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

to get all normal letters of the English alphabet, or

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

to get a hexadecimal digit, or

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

to get dates with leading zeros.

If the final value specified is not in the sequence that the magical increment would produce, the sequence goes until the next value would be longer than the final value specified.

As of Perl 5.26, the list-context range operator on strings works as expected in the scope of `"use feature 'unicode_strings"`. In previous versions, and outside the scope of that feature, it exhibits "The ''Unicode Bug"" in perlunicode: its behavior depends on the internal encoding of the range endpoint.

If the initial value specified isn't part of a magical increment sequence (that is, a non-empty string matching `/^[a-zA-Z]*[0-9]*\z/`), only the initial value will be returned. So the following will only return an alpha:

```
use charnames "greek";
my @greek_small =  ("\N{alpha}" .. "\N{omega}");
```

To get the 25 traditional lowercase Greek letters, including both sigmas, you could use this instead:

```
use charnames "greek";
my @greek_small =  map { chr } ( ord("\N{alpha}")
                                   ..
                                 ord("\N{omega}")
                               );
```

However, because there are *many* other lowercase Greek characters than just those, to match lowercase Greek characters in a regular expression, you could use the pattern `/(?:(?=\p{Greek})\p{Lower})+/` (or the experimental feature `/(?[ \p{Greek} & \p{Lower} ])+/`).

Because each operand is evaluated in integer form, `2.18 .. 3.14` will return two elements in list context.

```
@list = (2.18 .. 3.14); # same as @list = (2 .. 3);
```

Conditional Operator

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

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

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

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

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

```
($x_or_y ? $x : $y) = $z;
```

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

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

Really means this:

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

Rather than this:

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

That should probably be written more simply as:

```
$x += ($x % 2) ? 10 : 2;
```

Assignment Operators

"=" is the ordinary assignment operator.

Assignment operators work as in C. That is,

```
$x += 2;
```

is equivalent to

```
$x = $x + 2;
```

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

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

Although these are grouped by family, they all have the precedence of assignment. These combined assignment operators can only operate on scalars, whereas the ordinary assignment operator can assign to arrays, hashes, lists and even references. (See "Context" and "List value constructors" in perldata, and "Assigning to References" in perlref.)

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

```
($tmp = $global) =~ tr/13579/24680/;
```

Although as of 5.14, that can be also be accomplished this way:

```
use v5.14;
$tmp = ($global =~  tr/13579/24680/r);
```

Likewise,

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

is equivalent to

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

Similarly, a list assignment in list context produces the list of lvalues assigned to, and a list assignment in scalar context returns the number of elements produced by the expression on the right hand side of the assignment.

The three dotted bitwise assignment operators (`&.=` `|.=` `^.=`) are new in Perl 5.22 and experimental. See

"Bitwise String Operators".

Comma Operator

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

In list context, it's just the list argument separator, and inserts both its arguments into the list. These arguments are also evaluated from left to right.

The => operator (sometimes pronounced "fat comma") is a synonym for the comma except that it causes a word on its left to be interpreted as a string if it begins with a letter or underscore and is composed only of letters, digits and underscores. This includes operands that might otherwise be interpreted as operators, constants, single number v-strings or function calls. If in doubt about this behavior, the left operand can be quoted explicitly.

Otherwise, the => operator behaves exactly as the comma operator or list argument separator, according to context.

For example:

```
use constant FOO => "something";

my %h = ( FOO => 23 );
```

is equivalent to:

```
my %h = ("FOO", 23);
```

It is *NOT*:

```
my %h = ("something", 23);
```

The => operator is helpful in documenting the correspondence between keys and values in hashes, and other paired elements in lists.

```
%hash = ( $key => $value );
login( $username => $password );
```

The special quoting behavior ignores precedence, and hence may apply to *part* of the left operand:

```
print time.shift => "bbb";
```

That example prints something like `"1314363215shiftbbb"`, because the => implicitly quotes the `shift` immediately on its left, ignoring the fact that `time.shift` is the entire left operand.

List Operators (Rightward)

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

```
open HANDLE, "< :encoding(UTF-8)", "filename"
    or die "Can't open: $!\n";
```

However, some people find that code harder to read than writing it with parentheses:

```
open(HANDLE, "< :encoding(UTF-8)", "filename")
    or die "Can't open: $!\n";
```

in which case you might as well just use the more customary `"||"` operator:

```
open(HANDLE, "< :encoding(UTF-8)", "filename")
    || die "Can't open: $!\n";
```

See also discussion of list operators in "Terms and List Operators (Leftward)".

Logical Not

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

Logical And

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

if the left expression is true.

Logical or and Exclusive Or

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

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

This means that it short-circuits: the right expression is evaluated only if the left expression is false. Due to its precedence, you must be careful to avoid using it as replacement for the || operator. It usually works out better for flow control than in assignments:

```
$x = $y or $z;              # bug: this is wrong
($x = $y) or $z;            # really means this
$x = $y || $z;              # better written this way
```

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

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

Then again, you could always use parentheses.

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

There is no low precedence operator for defined-OR.

C Operators Missing From Perl

Here is what C has that Perl doesn't:

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

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

(TYPE) Type-casting operator.

Quote and Quote-like Operators

While we usually think of quotes as literal values, in Perl they function as operators, providing various kinds of interpolating and pattern matching capabilities. Perl provides customary quote characters for these behaviors, but also provides a way for you to choose your quote character for any of them. In the following table, a {} represents any pair of delimiters you choose.

```
Customary  Generic        Meaning        Interpolates
    ''       q{}          Literal             no
    ""      qq{}          Literal             yes
    ``      qx{}          Command             yes*
            qw{}         Word list            no
    //       m{}       Pattern match          yes*
            qr{}          Pattern             yes*
             s{}{}      Substitution          yes*
            tr{}{}    Transliteration         no (but see below)
             y{}{}    Transliteration         no (but see below)
    <<EOF                 here-doc            yes*

    * unless the delimiter is ''.
```

Non-bracketing delimiters use the same character fore and aft, but the four sorts of ASCII brackets (round, angle, square, curly) all nest, which means that

```
q{foo{bar}baz}
```

is the same as

```
'foo{bar}baz'
```

Note, however, that this does not always work for quoting Perl code:

```
$s = q{ if($x eq "}") ... }; # WRONG
```

is a syntax error. The Text::Balanced module (standard as of v5.8, and from CPAN before then) is

able to do this properly.

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

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

The cases where whitespace must be used are when the quoting character is a word character (meaning it matches `/\w/`):

```
q XfooX # Works: means the string 'foo'
qXfooX  # WRONG!
```

The following escape sequences are available in constructs that interpolate, and in transliterations:

```
Sequence     Note  Description
\t                 tab               (HT, TAB)
\n                 newline           (NL)
\r                 return            (CR)
\f                 form feed         (FF)
\b                 backspace         (BS)
\a                 alarm (bell)      (BEL)
\e                 escape            (ESC)
\x{263A}     [1,8] hex char          (example: SMILEY)
\x1b         [2,8] restricted range hex char (example: ESC)
\N{name}     [3]   named Unicode character or character sequence
\N{U+263D}   [4,8] Unicode character (example: FIRST QUARTER MOON)
\c[          [5]   control char      (example: chr(27))
\o{23072}    [6,8] octal char        (example: SMILEY)
\033         [7,8] restricted range octal char  (example: ESC)
```

[1] The result is the character specified by the hexadecimal number between the braces. See "[8]" below for details on which character.

Only hexadecimal digits are valid between the braces. If an invalid character is encountered, a warning will be issued and the invalid character and all subsequent characters (valid or invalid) within the braces will be discarded.

If there are no valid digits between the braces, the generated character is the NULL character (`\x{00}`). However, an explicit empty brace (`\x{}`) will not cause a warning (currently).

[2] The result is the character specified by the hexadecimal number in the range 0x00 to 0xFF. See "[8]" below for details on which character.

Only hexadecimal digits are valid following `\x`. When `\x` is followed by fewer than two valid digits, any valid digits will be zero-padded. This means that `\x7` will be interpreted as `\x07`, and a lone `"\x"` will be interpreted as `\x00`. Except at the end of a string, having fewer than two valid digits will result in a warning. Note that although the warning says the illegal character is ignored, it is only ignored as part of the escape and will still be used as the subsequent character in the string. For example:

```
Original    Result    Warns?
"\x7"       "\x07"    no
"\x"        "\x00"    no
"\x7q"      "\x07q"   yes
"\xq"       "\x00q"   yes
```

[3] The result is the Unicode character or character sequence given by *name*. See charnames.

[4] `\N{U+`*hexadecimal number*`}` means the Unicode character whose Unicode code point is *hexadecimal number*.

[5] The character following `\c` is mapped to some other character as shown in the table:

```
Sequence   Value
  \c@      chr(0)
  \cA      chr(1)
  \ca      chr(1)
  \cB      chr(2)
  \cb      chr(2)
  ...
  \cZ      chr(26)
  \cz      chr(26)
  \c[      chr(27)
                   # See below for chr(28)
  \c]      chr(29)
  \c^      chr(30)
  \c_      chr(31)
  \c?      chr(127) # (on ASCII platforms; see below for link to
                   #  EBCDIC discussion)
```

In other words, it's the character whose code point has had 64 xor'd with its uppercase. `\c?` is DELETE on ASCII platforms because `ord("?") ^ 64` is 127, and `\c@` is NULL because the ord of `"@"` is 64, so xor'ing 64 itself produces 0.

Also, `\c\`*X* yields `chr(28) . "`*X*`"` for any *X*, but cannot come at the end of a string, because the backslash would be parsed as escaping the end quote.

On ASCII platforms, the resulting characters from the list above are the complete set of ASCII controls. This isn't the case on EBCDIC platforms; see "OPERATOR DIFFERENCES" in perlebcdic for a full discussion of the differences between these for ASCII versus EBCDIC platforms.

Use of any other character following the `"c"` besides those listed above is discouraged, and as of Perl v5.20, the only characters actually allowed are the printable ASCII ones, minus the left brace `"{"`. What happens for any of the allowed other characters is that the value is derived by xor'ing with the seventh bit, which is 64, and a warning raised if enabled. Using the non-allowed characters generates a fatal error.

To get platform independent controls, you can use `\N{...}`.

[6] The result is the character specified by the octal number between the braces. See "[8]" below for details on which character.

If a character that isn't an octal digit is encountered, a warning is raised, and the value is based on the octal digits before it, discarding it and all following characters up to the closing brace. It is a fatal error if there are no octal digits at all.

[7] The result is the character specified by the three-digit octal number in the range 000 to 777 (but best to not use above 077, see next paragraph). See "[8]" below for details on which character.

Some contexts allow 2 or even 1 digit, but any usage without exactly three digits, the first being a zero, may give unintended results. (For example, in a regular expression it may be confused with a backreference; see "Octal escapes" in perlrebackslash.) Starting in Perl 5.14, you may use `\o{}` instead, which avoids all these problems. Otherwise, it is best to use this construct only for ordinals `\077` and below, remembering to pad to the left with zeros to make three digits. For larger ordinals, either use `\o{}`, or convert to something else, such as to hex and use `\N{U+}` (which is portable between platforms with different character sets) or `\x{}` instead.

[8] Several constructs above specify a character by a number. That number gives the character's position in the character set encoding (indexed from 0). This is called synonymously its ordinal, code position, or code point. Perl works on platforms that have a native encoding currently of either ASCII/Latin1 or EBCDIC, each of which allow specification of 256 characters. In general, if the number is 255 (0xFF, 0377) or below, Perl interprets this in the platform's native encoding. If the number is 256 (0x100, 0400) or above, Perl interprets it as a Unicode code point and the result is the corresponding Unicode character. For example `\x{50}` and `\o{120}` both are the number 80 in decimal, which is less than 256, so the number is interpreted in the native character set encoding. In ASCII the character in the 80th position (indexed from 0) is the letter `"P"`, and in EBCDIC it is the ampersand symbol `"&"`. `\x{100}` and `\o{400}` are both 256 in decimal, so the number is interpreted as a Unicode code

point no matter what the native encoding is. The name of the character in the 256th position (indexed by 0) in Unicode is `LATIN CAPITAL LETTER A WITH MACRON`.

An exception to the above rule is that `\N{U+`*hex number*`}` is always interpreted as a Unicode code point, so that `\N{U+0050}` is `"P"` even on EBCDIC platforms.

NOTE: Unlike C and other languages, Perl has no `\v` escape sequence for the vertical tab (VT, which is 11 in both ASCII and EBCDIC), but you may use `\N{VT}`, `\ck`, `\N{U+0b}`, or `\x0b`. (`\v` does have meaning in regular expression patterns in Perl, see perlre.)

The following escape sequences are available in constructs that interpolate, but not in transliterations.

```
\l          lowercase next character only
\u          titlecase (not uppercase!) next character only
\L          lowercase all characters till \E or end of string
\U          uppercase all characters till \E or end of string
\F          foldcase all characters till \E or end of string
\Q          quote (disable) pattern metacharacters till \E or
            end of string
\E          end either case modification or quoted section
            (whichever was last seen)
```

See ‘‘quotemeta’’ in perlfunc for the exact definition of characters that are quoted by `\Q`.

`\L`, `\U`, `\F`, and `\Q` can stack, in which case you need one `\E` for each. For example:

```
say"This \Qquoting \ubusiness \Uhere isn't quite\E done yet,\E is it?";
This quoting\ Business\ HERE\ ISN\'T\ QUITE\ done\ yet\, is it?
```

If a `use locale` form that includes `LC_CTYPE` is in effect (see perllocale), the case map used by `\l`, `\L`, `\u`, and `\U` is taken from the current locale. If Unicode (for example, `\N{}` or code points of 0x100 or beyond) is being used, the case map used by `\l`, `\L`, `\u`, and `\U` is as defined by Unicode. That means that case-mapping a single character can sometimes produce a sequence of several characters. Under `use locale`, `\F` produces the same results as `\L` for all locales but a UTF–8 one, where it instead uses the Unicode definition.

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

For constructs that do interpolate, variables beginning with `"$`‘‘ or ’’`@"` are interpolated. Subscripted variables such as `$a[3]` or `$href->{key}[0]` are also interpolated, as are array and hash slices. But method calls such as `$obj->meth` are not.

Interpolating an array or slice interpolates the elements in order, separated by the value of `$"`, so is equivalent to interpolating `join $", @array`. ‘‘Punctuation’’ arrays such as `@*` are usually interpolated only if the name is enclosed in braces `@{*}`, but the arrays `@_`, `@+`, and `@-` are interpolated even without braces.

For double-quoted strings, the quoting from `\Q` is applied after interpolation and escapes are processed.

```
"abc\Qfoo\tbar$s\Exyz"
```

is equivalent to

```
"abc" . quotemeta("foo\tbar$s") . "xyz"
```

For the pattern of regex operators (`qr//`, `m//` and `s///`), the quoting from `\Q` is applied after interpolation is processed, but before escapes are processed. This allows the pattern to match literally (except for `$` and `@`). For example, the following matches:

```
'\s\t' =~ /\Q\s\t/
```

Because `$` or `@` trigger interpolation, you'll need to use something like `/\Quser\E\@\Qhost/` to match them literally.

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

Apart from the behavior described above, Perl does not expand multiple levels of interpolation. In particular, contrary to the expectations of shell programmers, back-quotes do *NOT* interpolate within double quotes, nor do single quotes impede evaluation of variables when used within double quotes.

Regexp Quote-Like Operators

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

`qr/`*STRING*`/msixpodualn`

This operator quotes (and possibly compiles) its *STRING* as a regular expression. *STRING* is interpolated the same way as *PATTERN* in `m/`*PATTERN*`/`. If `"'"` is used as the delimiter, no variable interpolation is done. Returns a Perl value which may be used instead of the corresponding `/`*STRING*`/msixpodualn` expression. The returned value is a normalized version of the original pattern. It magically differs from a string containing the same characters: `ref(qr/x/)` returns "Regexp"; however, dereferencing it is not well defined (you currently get the normalized version of the original pattern, but this may change).

For example,

```
$rex = qr/my.STRING/is;
print $rex;                  # prints (?si-xm:my.STRING)
s/$rex/foo/;
```

is equivalent to

```
s/my.STRING/foo/is;
```

The result may be used as a subpattern in a match:

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

Since Perl may compile the pattern at the moment of execution of the `qr()` operator, using `qr()` may have speed advantages in some situations, notably if the result of `qr()` is used standalone:

```
sub match {
    my $patterns = shift;
    my @compiled = map qr/$_/i, @$patterns;
    grep {
        my $success = 0;
        foreach my $pat (@compiled) {
            $success = 1, last if /$pat/;
        }
        $success;
    } @_;
}
```

Precompilation of the pattern into an internal representation at the moment of `qr()` avoids the need to recompile the pattern every time a match `/$pat/` is attempted. (Perl has many other internal optimizations, but none would be triggered in the above example if we did not use `qr()` operator.)

Options (specified by the following modifiers) are:

```
m   Treat string as multiple lines.
s   Treat string as single line. (Make . match a newline)
i   Do case-insensitive pattern matching.
x   Use extended regular expressions; specifying two
    x's means \t and the SPACE character are ignored within
    square-bracketed character classes
p   When matching preserve a copy of the matched string so
    that ${^PREMATCH}, ${^MATCH}, ${^POSTMATCH} will be
    defined (ignored starting in v5.20) as these are always
    defined starting in that release
o   Compile pattern only once.
a   ASCII-restrict: Use ASCII for \d, \s, \w and [[:posix:]]
    character classes; specifying two a's adds the further
    restriction that no ASCII character will match a
    non-ASCII one under /i.
l   Use the current run-time locale's rules.
u   Use Unicode rules.
d   Use Unicode or native charset, as in 5.12 and earlier.
n   Non-capture mode. Don't let () fill in $1, $2, etc...
```

If a precompiled pattern is embedded in a larger pattern then the effect of `"msixpluadn"` will be propagated appropriately. The effect that the `/o` modifier has is not propagated, being restricted to those patterns explicitly using it.

The last four modifiers listed above, added in Perl 5.14, control the character set rules, but `/a` is the only one you are likely to want to specify explicitly; the other three are selected automatically by various pragmas.

See perlre for additional information on valid syntax for *STRING*, and for a detailed look at the semantics of regular expressions. In particular, all modifiers except the largely obsolete `/o` are further explained in "Modifiers" in perlre. `/o` is described in the next section.

`m/`*PATTERN*`/msixpodualngc`
`/`*PATTERN*`/msixpodualngc`

Searches a string for a pattern match, and in scalar context returns true if it succeeds, false if it fails. If no string is specified via the `=~` or `!~` operator, the `$_` string is searched. (The string specified with `=~` need not be an lvalue — it may be the result of an expression evaluation, but remember the `=~` binds rather tightly.) See also perlre.

Options are as described in `qr//` above; in addition, the following match process modifiers are available:

```
g  Match globally, i.e., find all occurrences.
c  Do not reset search position on a failed match when /g is
   in effect.
```

If `"/"` is the delimiter then the initial `m` is optional. With the `m` you can use any pair of non-whitespace (ASCII) characters as delimiters. This is particularly useful for matching path names that contain `"/"`, to avoid LTS (leaning toothpick syndrome). If `"?"` is the delimiter, then a match-only-once rule applies, described in `m?`*PATTERN*`?` below. If `"'"` (single quote) is the delimiter, no variable interpolation is performed on the *PATTERN*. When using a delimiter character valid in an identifier, whitespace is required after the `m`.

PATTERN may contain variables, which will be interpolated every time the pattern search is evaluated, except for when the delimiter is a single quote. (Note that `$(`, `$)`, and `$|` are not interpolated because they look like end-of-string tests.) Perl will not recompile the pattern unless an interpolated variable that it contains changes. You can force Perl to skip the test and never recompile by adding a `/o` (which stands for "once") after the trailing delimiter. Once upon a time, Perl would recompile regular expressions unnecessarily, and this modifier was useful to tell it not to do so, in the interests of speed. But now, the only reasons to use `/o` are one of:

1. The variables are thousands of characters long and you know that they don't change, and you need to wring out the last little bit of speed by having Perl skip testing for that. (There

is a maintenance penalty for doing this, as mentioning `/o` constitutes a promise that you won't change the variables in the pattern. If you do change them, Perl won't even notice.)

2. you want the pattern to use the initial values of the variables regardless of whether they change or not. (But there are saner ways of accomplishing this than using `/o`.)

3. If the pattern contains embedded code, such as

```
use re 'eval';
$code = 'foo(?{ $x })';
/$code/
```

then perl will recompile each time, even though the pattern string hasn't changed, to ensure that the current value of `$x` is seen each time. Use `/o` if you want to avoid this.

The bottom line is that using `/o` is almost never a good idea.

The empty pattern `//`

If the *PATTERN* evaluates to the empty string, the last *successfully* matched regular expression is used instead. In this case, only the `g` and `c` flags on the empty pattern are honored; the other flags are taken from the original pattern. If no match has previously succeeded, this will (silently) act instead as a genuine empty pattern (which will always match).

Note that it's possible to confuse Perl into thinking `//` (the empty regex) is really `//` (the defined-or operator). Perl is usually pretty good about this, but some pathological cases might trigger this, such as `$x///` (is that `($x) / (//)` or `$x // /`?) and `print $fh //` (`print $fh(//` or `print($fh //`?). In all of these examples, Perl will assume you meant defined-or. If you meant the empty regex, just use parentheses or spaces to disambiguate, or even prefix the empty regex with an `m` (so `//` becomes `m//`).

Matching in list context

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

Examples:

```
open(TTY, "+</dev/tty")
   || die "can't access /dev/tty: $!";

<TTY> =~ /^y/i && foo();       # do foo if desired

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

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

# poor man's grep
$arg = shift;
while (<>) {
   print if /$arg/o; # compile only once (no longer needed!)
}

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

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

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

In scalar context, each execution of `m//g` finds the next match, returning true if it matches, and

false if there is no further match. The position after the last match can be read or set using the `pos()` function; see "pos" in perlfunc. A failed match normally resets the search position to the beginning of the string, but you can avoid that by adding the `/c` modifier (for example, `m//gc`). Modifying the target string also resets the search position.

`\G` *assertion*

You can intermix `m//g` matches with `m/\G.../g`, where `\G` is a zero-width assertion that matches the exact position where the previous `m//g`, if any, left off. Without the `/g` modifier, the `\G` assertion still anchors at `pos()` as it was at the start of the operation (see "pos" in perlfunc), but the match is of course only attempted once. Using `\G` without `/g` on a target string that has not previously had a `/g` match applied to it is the same as using the `\A` assertion to match the beginning of the string. Note also that, currently, `\G` is only properly supported when anchored at the very beginning of the pattern.

Examples:

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

# scalar context
local $/ = "";
while ($paragraph = <>) {
    while ($paragraph =~ /\p{Ll}['")]*[.!?]+['")]*\s/g) {
        $sentences++;
    }
}
say $sentences;
```

Here's another way to check for sentences in a paragraph:

```
my $sentence_rx = qr{
    (?: (?<= ^ ) | (?<= \s ) )  # after start-of-string or
                                # whitespace
    \p{Lu}                      # capital letter
    .*?                         # a bunch of anything
    (?<= \S )                   # that ends in non-
                                # whitespace
    (?<! \b [DMS]r  )           # but isn't a common abbr.
    (?<! \b Mrs )
    (?<! \b Sra )
    (?<! \b St  )
    [.?!]                       # followed by a sentence
                                # ender
    (?= $ | \s )                # in front of end-of-string
                                # or whitespace
}sx;
local $/ = "";
while (my $paragraph = <>) {
   say "NEW PARAGRAPH";
   my $count = 0;
   while ($paragraph =~ /($sentence_rx)/g) {
       printf "\tgot sentence %d: <%s>\n", ++$count, $1;
   }
}
```

Here's how to use `m//gc` with `\G`:

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

The last example should print:

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

Notice that the final match matched `q` instead of `p`, which a match without the `\G` anchor would have done. Also note that the final match did not update `pos`. `pos` is only updated on a `/g` match. If the final match did indeed match `p`, it's a good bet that you're running a very old (pre–5.6.0) version of Perl.

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

```
 $_ = <<'EOL';
    $url = URI::URL->new( "http://example.com/" );
    die if $url eq "xXx";
 EOL

 LOOP: {
     print(" digits"),        redo LOOP if /\G\d+\b[,.;]?\s*/gc;
     print(" lowercase"),     redo LOOP
                                if /\G\p{Ll}+\b[,.;]?\s*/gc;
     print(" UPPERCASE"),     redo LOOP
                                if /\G\p{Lu}+\b[,.;]?\s*/gc;
     print(" Capitalized"),   redo LOOP
                               if /\G\p{Lu}\p{Ll}+\b[,.;]?\s*/gc;
     print(" MiXeD"),         redo LOOP if /\G\pL+\b[,.;]?\s*/gc;
     print(" alphanumeric"), redo LOOP
                             if /\G[\p{Alpha}\pN]+\b[,.;]?\s*/gc;
     print(" line-noise"),    redo LOOP if /\G\W+/gc;
     print ". That's all!\n";
 }
```

Here is the output (split into several lines):

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

`m?`*PATTERN*`?msixpodualngc`

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

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

Another example switched the first "latin1" encoding it finds to "utf8" in a pod file:

```
s//utf8/ if m? ^ =encoding \h+ \K latin1 ?x;
```

The match-once behavior is controlled by the match delimiter being `?`; with any other delimiter this is the normal `m//` operator.

In the past, the leading `m` in `m?`*PATTERN*`?` was optional, but omitting it would produce a deprecation warning. As of v5.22.0, omitting it produces a syntax error. If you encounter this construct in older code, you can just add `m`.

`s/`*PATTERN*`/`*REPLACEMENT*`/msixpodualngcer`

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

If the `/r` (non-destructive) option is used then it runs the substitution on a copy of the string and instead of returning the number of substitutions, it returns the copy whether or not a substitution occurred. The original string is never changed when `/r` is used. The copy will always be a plain string, even if the input is an object or a tied variable.

If no string is specified via the `=~` or `!~` operator, the `$_` variable is searched and modified. Unless the `/r` option is used, the string specified must be a scalar variable, an array element, a hash element, or an assignment to one of those; that is, some sort of scalar lvalue.

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

Options are as with `m//` with the addition of the following replacement specific options:

```
e   Evaluate the right side as an expression.
ee  Evaluate the right side as a string then eval the
    result.
r   Return substitution and leave the original string
    untouched.
```

Any non-whitespace delimiter may replace the slashes. Add space after the `s` when using a character allowed in identifiers. If single quotes are used, no interpretation is done on the replacement string (the `/e` modifier overrides this, however). Note that Perl treats backticks as normal delimiters; the replacement text is not evaluated as a command. If the *PATTERN* is delimited by bracketing quotes, the *REPLACEMENT* has its own pair of quotes, which may or may not be bracketing quotes, for example, `s(foo)(bar)` or `s<foo>/bar/`. A `/e` will cause the replacement portion to be treated as a full-fledged Perl expression and evaluated right then and there. It is, however, syntax checked at compile-time. A second `e` modifier will cause the replacement portion to be `eval`ed before being run as a Perl expression.

Examples:

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

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

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

```
($foo = $bar) =~ s/this/that/;      # copy first, then
                                    # change
($foo = "$bar") =~ s/this/that/;    # convert to string,
                                    # copy, then change
$foo = $bar =~ s/this/that/r;       # Same as above using /r
$foo = $bar =~ s/this/that/r
            =~ s/that/the other/r;  # Chained substitutes
                                    # using /r
@foo = map { s/this/that/r } @bar   # /r is very useful in
                                    # maps

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

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

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

$_ = 'abc123xyz';
$x = s/abc/def/r;           # $x is 'def123xyz' and
                            # $_ remains 'abc123xyz'.

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

# Add one to the value of any numbers in the string
s/(\d+)/1 + $1/eg;

# Titlecase words in the last 30 characters only
substr($str, -30) =~ s/\b(\p{Alpha}+)\b/\u\L$1/g;

# This will expand any embedded scalar variable
# (including lexicals) in $_ : First $1 is interpolated
# to the variable name, and then evaluated
s/(\$\w+)/$1/eeg;

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

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

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

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

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

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

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

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

Quote-Like Operators

`q/`*STRING*`/`
`'`*STRING*`'`

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

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

`qq/`*STRING*`/`
"*STRING*"

A double-quoted, interpolated string.

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

`qx/`*STRING*`/`
`` `*STRING*` ``

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

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

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

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

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

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

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

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

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

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

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

The STDIN filehandle used by the command is inherited from Perl's STDIN. For example:

```
open(SPLAT, "stuff")   || die "can't open stuff: $!";
open(STDIN, "<&SPLAT") || die "can't dupe SPLAT: $!";
print STDOUT `sort`;
```

will print the sorted contents of the file named *"stuff"*.

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

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

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

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

Perl will attempt to flush all files opened for output before starting the child process, but this may not be supported on some platforms (see perlport). To be safe, you may need to set `$|` (`$AUTOFLUSH` in `English`) or call the `autoflush()` method of `IO::Handle` on any open handles.

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

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

Like `system`, backticks put the child process exit code in `$?`. If you'd like to manually inspect failure, you can check all possible failure modes by inspecting `$?` like this:

```
if ($? == -1) {
    print "failed to execute: $!\n";
}
elsif ($? & 127) {
    printf "child died with signal %d, %s coredump\n",
        ($? & 127),  ($? & 128) ? 'with' : 'without';
}
else {
    printf "child exited with value %d\n", $? >> 8;
}
```

Use the open pragma to control the I/O layers used when reading the output of the command, for example:

```
use open IN => ":encoding(UTF-8)";
my $x = `cmd-producing-utf-8`;
```

See "I/O Operators" for more discussion.

`qw/`*`STRING`*`/`

Evaluates to a list of the words extracted out of *STRING*, using embedded whitespace as the word delimiters. It can be understood as being roughly equivalent to:

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

the differences being that it generates a real list at compile time, and in scalar context it returns the last element in the list. So this expression:

```
qw(foo bar baz)
```

is semantically equivalent to the list:

```
"foo", "bar", "baz"
```

Some frequently seen examples:

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

A common mistake is to try to separate the words with commas or to put comments into a multi-line qw-string. For this reason, the `use warnings` pragma and the **-w** switch (that is, the `$^W` variable) produces warnings if the *STRING* contains the `","` or the `"#"` character.

tr/*SEARCHLIST*/*REPLACEMENTLIST*/cdsr
y/*SEARCHLIST*/*REPLACEMENTLIST*/cdsr

Transliterates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced or deleted. If no string is specified via the =~ or !~ operator, the $_ string is transliterated.

If the /r (non-destructive) option is present, a new copy of the string is made and its characters transliterated, and this copy is returned no matter whether it was modified or not: the original string is always left unchanged. The new copy is always a plain string, even if the input string is an object or a tied variable.

Unless the /r option is used, the string specified with =~ must be a scalar variable, an array element, a hash element, or an assignment to one of those; in other words, an lvalue.

A character range may be specified with a hyphen, so `tr/A-J/0-9/` does the same replacement as `tr/ACEGIBDFHJ/0246813579/`. For **sed** devotees, `y` is provided as a synonym for `tr`. If the *SEARCHLIST* is delimited by bracketing quotes, the *REPLACEMENTLIST* must have its own pair of quotes, which may or may not be bracketing quotes; for example, `tr[aeiouy][yuoiea]` or `tr(+\-*/)/ABCD/`.

Characters may be literals or any of the escape sequences accepted in double-quoted strings. But there is no variable interpolation, so `"$"` and `"@"` are treated as literals. A hyphen at the beginning or end, or preceded by a backslash is considered a literal. Escape sequence details are in the table near the beginning of this section.

Note that `tr` does **not** do regular expression character classes such as `\d` or `\pL`. The `tr` operator is not equivalent to the `tr(1)` utility. `tr[a-z][A-Z]` will uppercase the 26 letters "a" through "z", but for case changing not confined to ASCII, use `lc`, `uc`, `lcfirst`, `ucfirst` (all documented in perlfunc), or the substitution operator `s/PATTERN/REPLACEMENT/` (with `\U`, `\u`, `\L`, and `\l` string-interpolation escapes in the *REPLACEMENT* portion).

Most ranges are unportable between character sets, but certain ones signal Perl to do special handling to make them portable. There are two classes of portable ranges. The first are any subsets of the ranges `A-Z`, `a-z`, and `0-9`, when expressed as literal characters.

```
tr/h-k/H-K/
```

capitalizes the letters `"h"`, `"i"`, `"j"`, and `"k"` and nothing else, no matter what the platform's character set is. In contrast, all of

```
tr/\x68-\x6B/\x48-\x4B/
tr/h-\x6B/H-\x4B/
tr/\x68-k/\x48-K/
```

do the same capitalizations as the previous example when run on ASCII platforms, but something completely different on EBCDIC ones.

The second class of portable ranges is invoked when one or both of the range's end points are expressed as `\N{...}`

```
$string =~ tr/\N{U+20}-\N{U+7E}//d;
```

removes from `$string` all the platform's characters which are equivalent to any of Unicode U+0020,

U+0021, ... U+007D, U+007E. This is a portable range, and has the same effect on every platform it is run on. It turns out that in this example, these are the ASCII printable characters. So after this is run, `$string` has only controls and characters which have no ASCII equivalents.

But, even for portable ranges, it is not generally obvious what is included without having to look things up. A sound principle is to use only ranges that begin from and end at either ASCII alphabetics of equal case (`b-e`, `B-E`), or digits (`1-4`). Anything else is unclear (and unportable unless `\N{...}` is used). If in doubt, spell out the character sets in full.

Options:

```
c   Complement the SEARCHLIST.
d   Delete found but unreplaced characters.
s   Squash duplicate replaced characters.
r   Return the modified string and leave the original string
    untouched.
```

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

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

Examples:

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

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

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

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

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

($HOST = $host) =~ tr/a-z/A-Z/;
 $HOST = $host  =~ tr/a-z/A-Z/r;   # same thing

$HOST = $host =~ tr/a-z/A-Z/r    # chained with s///r
              =~ s/:/ -p/r;

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

@stripped = map tr/a-zA-Z/ /csr, @original;
                            # /r with map

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

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

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

will transliterate any A to X.

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

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

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

<<EOF

A line-oriented form of quoting is based on the shell "here-document" syntax. Following a << you specify a string to terminate the quoted material, and all lines following the current line down to the terminating string are the value of the item.

Prefixing the terminating string with a ~ specifies that you want to use "Indented Here-docs" (see below).

The terminating string may be either an identifier (a word), or some quoted text. An unquoted identifier works like double quotes. There may not be a space between the << and the identifier, unless the identifier is explicitly quoted. (If you put a space it will be treated as a null identifier, which is valid, and matches the first empty line.) The terminating string must appear by itself (unquoted and with no surrounding whitespace) on the terminating line.

If the terminating string is quoted, the type of quotes used determine the treatment of the text.

Double Quotes

Double quotes indicate that the text will be interpolated using exactly the same rules as normal double quoted strings.

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

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

Single Quotes

Single quotes indicate the text is to be treated literally with no interpolation of its content. This is similar to single quoted strings except that backslashes have no special meaning, with \\ being treated as two backslashes and not one as they would in every other quoting construct.

Just as in the shell, a backslashed bareword following the << means the same thing as a single-quoted string does:

```
        $cost = <<'VISTA';  # hasta la ...
    That'll be $10 please, ma'am.
    VISTA

        $cost = <<\VISTA;   # Same thing!
    That'll be $10 please, ma'am.
    VISTA
```

This is the only form of quoting in perl where there is no need to worry about escaping content, something that code generators can and do make good use of.

Backticks

The content of the here doc is treated just as it would be if the string were embedded in backticks. Thus the content is interpolated as though it were double quoted and then executed via the shell, with the results of the execution returned.

```
       print << `EOC`; # execute command and get results
    echo hi there
    EOC
```

Indented Here-docs

The here-doc modifier ~ allows you to indent your here-docs to make the code more readable:

```
    if ($some_var) {
      print <<~EOF;
        This is a here-doc
        EOF
    }
```

This will print...

```
    This is a here-doc
```

...with no leading whitespace.

The delimiter is used to determine the **exact** whitespace to remove from the beginning of each line. All lines **must** have at least the same starting whitespace (except lines only containing a newline) or perl will croak. Tabs and spaces can be mixed, but are matched exactly. One tab will not be equal to 8 spaces!

Additional beginning whitespace (beyond what preceded the delimiter) will be preserved:

```
    print <<~EOF;
      This text is not indented
        This text is indented with two spaces
                This text is indented with two tabs
      EOF
```

Finally, the modifier may be used with all of the forms mentioned above:

```
    <<~\EOF;
    <<~'EOF'
    <<~"EOF"
    <<~`EOF`
```

And whitespace may be used between the ~ and quoted delimiters:

```
    <<~ 'EOF'; # ... "EOF", `EOF`
```

It is possible to stack multiple here-docs in a row:

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

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

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

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

If you want to remove the line terminator from your here-docs, use `chomp()`.

```
chomp($string = <<'END');
This is a string.
END
```

If you want your here-docs to be indented with the rest of the code, you'll need to remove leading whitespace from each line manually:

```
($quote = <<'FINIS') =~ s/^\s+//gm;
   The Road goes ever on and on,
   down from the door where it began.
FINIS
```

If you use a here-doc within a delimited construct, such as in `s///eg`, the quoted material must still come on the line following the `<<FOO` marker, which means it may be inside the delimited construct:

```
s/this/<<E . 'that'
the other
E
 . 'more '/eg;
```

It works this way as of Perl 5.18. Historically, it was inconsistent, and you would have to write

```
s/this/<<E . 'that'
 . 'more '/eg;
the other
E
```

outside of string evals.

Additionally, quoting rules for the end-of-string identifier are unrelated to Perl's quoting rules. `q()`, `qq()`, and the like are not supported in place of `''` and `""`, and the only interpolation is for backslashing the quoting character:

```
print << "abc\"def";
testing...
abc"def
```

Finally, quoted strings cannot span multiple lines. The general rule is that the identifier must be a string literal. Stick with that, and you should be safe.

Gory details of parsing quoted constructs

When presented with something that might have several different interpretations, Perl uses the **DWIM** (that's "Do What I Mean") principle to pick the most probable interpretation. This strategy is so successful that Perl programmers often do not suspect the ambivalence of what they write. But from time to time, Perl's notions differ substantially from what the author honestly meant.

This section hopes to clarify how Perl handles quoted constructs. Although the most common reason to learn this is to unravel labyrinthine regular expressions, because the initial steps of parsing are the same for all quoting operators, they are all discussed together.

The most important Perl parsing rule is the first one discussed below: when processing a quoted construct, Perl first finds the end of that construct, then interprets its contents. If you understand this rule, you may skip the rest of this section on the first reading. The other rules are likely to contradict the user's expectations much less frequently than this first one.

Some passes discussed below are performed concurrently, but because their results are the same, we consider them individually. For different quoting constructs, Perl performs different numbers of passes, from one to four, but these passes are always performed in the same order.

Finding the end

The first pass is finding the end of the quoted construct. This results in saving to a safe location a copy of the text (between the starting and ending delimiters), normalized as necessary to avoid needing to know what the original delimiters were.

If the construct is a here-doc, the ending delimiter is a line that has a terminating string as the content. Therefore `<<EOF` is terminated by `EOF` immediately followed by `"\n"` and starting from the first column of the terminating line. When searching for the terminating line of a here-doc, nothing is skipped. In other words, lines after the here-doc syntax are compared with the terminating string line by line.

For the constructs except here-docs, single characters are used as starting and ending delimiters. If the starting delimiter is an opening punctuation (that is `(`, `[`, `{`, or `<`), the ending delimiter is the corresponding closing punctuation (that is `)`, `]`, `}`, or `>`). If the starting delimiter is an unpaired character like `/` or a closing punctuation, the ending delimiter is the same as the starting delimiter.

Therefore a / terminates a qq// construct, while a] terminates both qq[] and qq]] constructs.

When searching for single-character delimiters, escaped delimiters and \\ are skipped. For example, while searching for terminating /, combinations of \\ and \/ are skipped. If the delimiters are bracketing, nested pairs are also skipped. For example, while searching for a closing] paired with the opening [, combinations of \\, \], and \[are all skipped, and nested [and] are skipped as well. However, when backslashes are used as the delimiters (like qq\\ and tr\\\), nothing is skipped. During the search for the end, backslashes that escape delimiters or other backslashes are removed (exactly speaking, they are not copied to the safe location).

For constructs with three-part delimiters (s///, y///, and tr///), the search is repeated once more. If the first delimiter is not an opening punctuation, the three delimiters must be the same, such as s!!! and tr)))), in which case the second delimiter terminates the left part and starts the right part at once. If the left part is delimited by bracketing punctuation (that is (), [], {}, or <>), the right part needs another pair of delimiters such as s(){} and tr[]//. In these cases, whitespace and comments are allowed between the two parts, although the comment must follow at least one whitespace character; otherwise a character expected as the start of the comment may be regarded as the starting delimiter of the right part.

During this search no attention is paid to the semantics of the construct. Thus:

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

or:

```
m/
  bar        # NOT a comment, this slash / terminated m//!
 /x
```

do not form legal quoted expressions. The quoted part ends on the first " and /, and the rest happens to be a syntax error. Because the slash that terminated m// was followed by a SPACE, the example above is not m//x, but rather m// with no /x modifier. So the embedded # is interpreted as a literal #.

Also no attention is paid to \c\ (multichar control char syntax) during this search. Thus the second \ in qq/\c\/ is interpreted as a part of \/, and the following / is not recognized as a delimiter. Instead, use \034 or \x1c at the end of quoted constructs.

Interpolation

The next step is interpolation in the text obtained, which is now delimiter-independent. There are multiple cases.

<<'EOF'

No interpolation is performed. Note that the combination \\ is left intact, since escaped delimiters are not available for here-docs.

m'', the pattern of s'''

No interpolation is performed at this stage. Any backslashed sequences including \\ are treated at the stage to "parsing regular expressions".

'', q//, tr''', y''', the replacement of s'''

The only interpolation is removal of \ from pairs of \\. Therefore "-" in tr''' and y''' is treated literally as a hyphen and no character range is available. \1 in the replacement of s''' does not work as $1.

tr///, y///

No variable interpolation occurs. String modifying combinations for case and quoting such as \Q, \U, and \E are not recognized. The other escape sequences such as \200 and \t and backslashed characters such as \\ and \- are converted to appropriate literals. The character "-" is treated specially and therefore \- is treated as a literal "-".

``'', ``, qq//, qx//, <file*glob>, <<``EOF''

\Q, \U, \u, \L, \l, \F (possibly paired with \E) are converted to corresponding Perl constructs. Thus, "$foo\Qbaz$bar" is converted to $foo . (quotemeta("baz" . $bar)) internally. The other escape sequences such as \200 and \t and backslashed characters such as \\ and \- are replaced with appropriate

expansions.

Let it be stressed that *whatever falls between* `\Q` *and* `\E` is interpolated in the usual way. Something like `"\Q\\E"` has no `\E` inside. Instead, it has `\Q`, `\\`, and `E`, so the result is the same as for `"\\\\E"`. As a general rule, backslashes between `\Q` and `\E` may lead to counterintuitive results. So, `"\Q\t\E"` is converted to `quotemeta("\t")`, which is the same as `"\\\t"` (since TAB is not alphanumeric). Note also that:

```
$str = '\t';
return "\Q$str";
```

may be closer to the conjectural *intention* of the writer of `"\Q\t\E"`.

Interpolated scalars and arrays are converted internally to the `join` and `"."` catenation operations. Thus, `"$foo XXX '@arr'"` becomes:

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

All operations above are performed simultaneously, left to right.

Because the result of `"\Q` *STRING* `\E"` has all metacharacters quoted, there is no way to insert a literal `$` or `@` inside a `\Q\E` pair. If protected by `\`, `$` will be quoted to become `"\\\$"`; if not, it is interpreted as the start of an interpolated scalar.

Note also that the interpolation code needs to make a decision on where the interpolated scalar ends. For instance, whether `"a $x -> {c}"` really means:

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

or:

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

Most of the time, the longest possible text that does not include spaces between components and which contains matching braces or brackets. because the outcome may be determined by voting based on heuristic estimators, the result is not strictly predictable. Fortunately, it's usually correct for ambiguous cases.

the replacement of `s///`
: Processing of `\Q`, `\U`, `\u`, `\L`, `\l`, `\F` and interpolation happens as with `qq//` constructs.

  It is at this step that `\1` is begrudgingly converted to `$1` in the replacement text of `s///`, in order to correct the incorrigible *sed* hackers who haven't picked up the saner idiom yet. A warning is emitted if the `use warnings` pragma or the **-w** command-line flag (that is, the `$^W` variable) was set.

RE in `m?RE?`, `/RE/`, `m/RE/`, `s/RE/foo/`,
: Processing of `\Q`, `\U`, `\u`, `\L`, `\l`, `\F`, `\E`, and interpolation happens (almost) as with `qq//` constructs.

  Processing of `\N{...}` is also done here, and compiled into an intermediate form for the regex compiler. (This is because, as mentioned below, the regex compilation may be done at execution time, and `\N{...}` is a compile-time construct.)

  However any other combinations of `\` followed by a character are not substituted but only skipped, in order to parse them as regular expressions at the following step. As `\c` is skipped at this step, `@` of `\c@` in RE is possibly treated as an array symbol (for example `@foo`), even though the same text in `qq//` gives interpolation of `\c@`.

  Code blocks such as `(?{BLOCK})` are handled by temporarily passing control back to the perl parser, in a similar way that an interpolated array subscript expression such as `"foo$array[1+f("[xyz")]bar"` would be.

  Moreover, inside `(?{BLOCK})`, `(?# comment )`, and a `#`-comment in a `/x`-regular expression, no processing is performed whatsoever. This is the first step at which the presence of the `/x` modifier is relevant.

  Interpolation in patterns has several quirks: `$|`, `$(`, `$)`, `@+` and `@-` are not interpolated, and constructs `$var[SOMETHING]` are voted (by several different estimators) to be either an array

element or `$var` followed by an RE alternative. This is where the notation `${arr[$bar]}` comes handy: `/${arr[0-9]}/` is interpreted as array element `-9`, not as a regular expression from the variable `$arr` followed by a digit, which would be the interpretation of `/$arr[0-9]/`. Since voting among different estimators may occur, the result is not predictable.

The lack of processing of `\\` creates specific restrictions on the post-processed text. If the delimiter is `/`, one cannot get the combination `\/` into the result of this step. `/` will finish the regular expression, `\/` will be stripped to `/` on the previous step, and `\\/` will be left as is. Because `/` is equivalent to `\/` inside a regular expression, this does not matter unless the delimiter happens to be character special to the RE engine, such as in `s*foo*bar*`, `m[foo]`, or `m?foo?`; or an alphanumeric char, as in:

```
m m ^ a \s* b mmx;
```

In the RE above, which is intentionally obfuscated for illustration, the delimiter is `m`, the modifier is `mx`, and after delimiter-removal the RE is the same as for `m/ ^ a \s* b /mx`. There's more than one reason you're encouraged to restrict your delimiters to non-alphanumeric, non-whitespace choices.

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

parsing regular expressions

Previous steps were performed during the compilation of Perl code, but this one happens at run time, although it may be optimized to be calculated at compile time if appropriate. After preprocessing described above, and possibly after evaluation if concatenation, joining, casing translation, or metaquoting are involved, the resulting *string* is passed to the RE engine for compilation.

Whatever happens in the RE engine might be better discussed in perlre, but for the sake of continuity, we shall do so here.

This is another step where the presence of the `/x` modifier is relevant. The RE engine scans the string from left to right and converts it into a finite automaton.

Backslashed characters are either replaced with corresponding literal strings (as with `\{`), or else they generate special nodes in the finite automaton (as with `\b`). Characters special to the RE engine (such as `|`) generate corresponding nodes or groups of nodes. `(?#...)` comments are ignored. All the rest is either converted to literal strings to match, or else is ignored (as is whitespace and `#`-style comments if `/x` is present).

Parsing of the bracketed character class construct, `[...]`, is rather different than the rule used for the rest of the pattern. The terminator of this construct is found using the same rules as for finding the terminator of a `{}`-delimited construct, the only exception being that `]` immediately following `[` is treated as though preceded by a backslash.

The terminator of runtime `(?{...})` is found by temporarily switching control to the perl parser, which should stop at the point where the logically balancing terminating `}` is found.

It is possible to inspect both the string given to RE engine and the resulting finite automaton. See the arguments `debug`/`debugcolor` in the `use re` pragma, as well as Perl's **-Dr** command-line switch documented in "Command Switches" in perlrun.

Optimization of regular expressions

This step is listed for completeness only. Since it does not change semantics, details of this step are not documented and are subject to change without notice. This step is performed over the finite automaton that was generated during the previous pass.

It is at this stage that `split()` silently optimizes `/^/` to mean `/^/m`.

I/O Operators

There are several I/O operators you should know about.

A string enclosed by backticks (grave accents) first undergoes double-quote interpolation. It is then interpreted as an external command, and the output of that command is the value of the backtick string, like in a shell. In scalar context, a single string consisting of all output is returned. In list context, a list of values is returned, one per line of output. (You can set `$/` to use a different line terminator.) The command is executed each time the pseudo-literal is evaluated. The status value of the command is returned in `$?`

(see perlvar for the interpretation of `$?`). Unlike in **csh**, no translation is done on the return data — newlines remain newlines. Unlike in any of the shells, single quotes do not hide variable names in the command from interpretation. To pass a literal dollar-sign through to the shell you need to hide it with a backslash. The generalized form of backticks is `qx//`. (Because backticks always undergo shell expansion as well, see perlsec for security concerns.)

In scalar context, evaluating a filehandle in angle brackets yields the next line from that file (the newline, if any, included), or `undef` at end-of-file or on error. When `$/` is set to `undef` (sometimes known as file-slurp mode) and the file is empty, it returns `''` the first time, followed by `undef` subsequently.

Ordinarily you must assign the returned value to a variable, but there is one situation where an automatic assignment happens. If and only if the input symbol is the only thing inside the conditional of a `while` statement (even if disguised as a `for(;;)` loop), the value is automatically assigned to the global variable `$_`, destroying whatever was there previously. (This may seem like an odd thing to you, but you'll use the construct in almost every Perl script you write.) The `$_` variable is not implicitly localized. You'll have to put a `local $_;` before the loop if you want that to happen.

The following lines are equivalent:

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

This also behaves similarly, but assigns to a lexical variable instead of to `$_`:

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

In these loop constructs, the assigned value (whether assignment is automatic or explicit) is then tested to see whether it is defined. The defined test avoids problems where the line has a string value that would be treated as false by Perl; for example a "" or a `"0"` with no trailing newline. If you really mean for such values to terminate the loop, they should be tested for explicitly:

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

In other boolean contexts, *<FILEHANDLE>* without an explicit `defined` test or comparison elicits a warning if the `use warnings` pragma or the **-w** command-line switch (the `$^W` variable) is in effect.

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

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

<FILEHANDLE> may also be spelled `readline(*FILEHANDLE)`. See "readline" in perlfunc.

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

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

is equivalent to the following Perl-like pseudo code:

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

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

Since the null filehandle uses the two argument form of "open" in perlfunc it interprets special characters, so if you have a script like this:

```
while (<>) {
    print;
}
```

and call it with `perl dangerous.pl 'rm -rfv *|'`, it actually opens a pipe, executes the `rm` command and reads `rm`'s output from that pipe. If you want all items in `@ARGV` to be interpreted as file names, you can use the module `ARGV::readonly` from CPAN, or use the double bracket:

```
while (<<>>) {
    print;
}
```

Using double angle brackets inside of a while causes the open to use the three argument form (with the second argument being <), so all arguments in `ARGV` are treated as literal filenames (including `"-"`). (Note that for convenience, if you use `<<>>` and if `@ARGV` is empty, it will still read from the standard input.)

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

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

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

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

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

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

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

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

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

If what the angle brackets contain is a simple scalar variable (for example, `$foo`), then that variable contains the name of the filehandle to input from, or its typeglob, or a reference to the same. For example:

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

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

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

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

is roughly equivalent to:

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

except that the globbing is actually done internally using the standard `File::Glob` extension. Of course, the shortest way to do the above is:

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

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

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

than

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

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

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

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

Constant Folding

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

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

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

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

the compiler precomputes the number which that expression represents so that the interpreter won't have to.

No-ops

Perl doesn't officially have a no-op operator, but the bare constants `0` and `1` are special-cased not to produce a warning in void context, so you can for example safely do

```
1 while foo();
```

Bitwise String Operators

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

If the operands to a binary bitwise op are strings of different sizes, **|** and **^** ops act as though the shorter operand had additional zero bits on the right, while the **&** op acts as though the longer operand were truncated to the length of the shorter. The granularity for such extension or truncation is one or more bytes.

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

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

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

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

This somewhat unpredictable behavior can be avoided with the experimental "bitwise" feature, new in Perl 5.22. You can enable it via `use feature 'bitwise'`. By default, it will warn unless the `"experimental::bitwise"` warnings category has been disabled. (`use experimental 'bitwise'` will enable the feature and disable the warning.) Under this feature, the four standard bitwise operators (`~ | & ^`) are always numeric. Adding a dot after each operator (`~. |. &. ^.`) forces it to treat its operands as strings:

```
use experimental "bitwise";
$foo =  150  |  105;        # yields 255  (0x96 | 0x69 is 0xFF)
$foo = '150' |  105;        # yields 255
$foo =  150  | '105';       # yields 255
$foo = '150' | '105';       # yields 255
$foo =  150  |. 105;        # yields string '155'
$foo = '150' |. 105;        # yields string '155'
$foo =  150  |.'105';       # yields string '155'
$foo = '150' |.'105';       # yields string '155'

$baz = $foo &  $bar;        # both operands numeric
$biz = $foo ^. $bar;        # both operands stringy
```

The assignment variants of these operators (`&= |= ^= &.= |.= ^.=`) behave likewise under the feature.

The behavior of these operators is problematic (and subject to change) if either or both of the strings are encoded in UTF-8 (see "Byte and Character Semantics" in perlunicode.

See "vec" in perlfunc for information on how to manipulate individual bits in a bit vector.

Integer Arithmetic

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

```
use integer;
```

you may tell the compiler to use integer operations (see integer for a detailed explanation) from here to the end of the enclosing BLOCK. An inner BLOCK may countermand this by saying

```
no integer;
```

which lasts until the end of that BLOCK. Note that this doesn't mean everything is an integer, merely that Perl will use integer operations for arithmetic, comparison, and bitwise operators. For example, even under `use integer`, if you take the `sqrt(2)`, you'll still get `1.4142135623731` or so.

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

Floating-point Arithmetic

While `use integer` provides integer-only arithmetic, there is no analogous mechanism to provide automatic rounding or truncation to a certain number of decimal places. For rounding to a certain number of digits, `sprintf()` or `printf()` is usually the easiest route. See perlfaq4.

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

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

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

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

The POSIX module (part of the standard perl distribution) implements `ceil()`, `floor()`, and other mathematical and trigonometric functions. The `Math::Complex` module (part of the standard perl distribution) defines mathematical functions that work on both the reals and the imaginary numbers. `Math::Complex` is not as efficient as POSIX, but POSIX can't work with complex numbers.

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

Bigger Numbers

The standard `Math::BigInt`, `Math::BigRat`, and `Math::BigFloat` modules, along with the `bignum`, `bigint`, and `bigrat` pragmas, provide variable-precision arithmetic and overloaded operators, although they're currently pretty slow. At the cost of some space and considerable speed, they avoid the normal pitfalls associated with limited-precision representations.

```
    use 5.010;
    use bigint;  # easy interface to Math::BigInt
    $x = 123456789123456789;
    say $x * $x;
+15241578780673678515622620750190521
```

Or with rationals:

```
use 5.010;
use bigrat;
$x = 3/22;
$y = 4/6;
say "x/y is ", $x/$y;
say "x*y is ", $x*$y;
x/y is 9/44
x*y is 1/11
```

Several modules let you calculate with unlimited or fixed precision (bound only by memory and CPU time). There are also some non-standard modules that provide faster implementations via external C libraries.

Here is a short, but incomplete summary:

```
Math::String           treat string sequences like numbers
Math::FixedPrecision   calculate with a fixed precision
Math::Currency         for currency calculations
Bit::Vector            manipulate bit vectors fast (uses C)
Math::BigIntFast       Bit::Vector wrapper for big numbers
Math::Pari             provides access to the Pari C library
Math::Cephes           uses the external Cephes C library (no
                       big numbers)
Math::Cephes::Fraction fractions via the Cephes library
Math::GMP              another one using an external C library
Math::GMPz             an alternative interface to libgmp's big ints
Math::GMPq             an interface to libgmp's fraction numbers
Math::GMPf             an interface to libgmp's floating point numbers
```

Choose wisely.

NAME

perlsub – Perl subroutines

SYNOPSIS

To declare subroutines:

```
sub NAME;                       # A "forward" declaration.
sub NAME(PROTO);                #  ditto, but with prototypes
sub NAME : ATTRS;               #  with attributes
sub NAME(PROTO) : ATTRS;        #  with attributes and prototypes

sub NAME BLOCK                  # A declaration and a definition.
sub NAME(PROTO) BLOCK           #  ditto, but with prototypes
sub NAME(SIG) BLOCK             #  with a signature instead
sub NAME : ATTRS BLOCK          #  with attributes
sub NAME(PROTO) : ATTRS BLOCK #  with prototypes and attributes
sub NAME(SIG) : ATTRS BLOCK   #  with a signature and attributes
```

To define an anonymous subroutine at runtime:

```
$subref = sub BLOCK;                  # no proto
$subref = sub (PROTO) BLOCK;          # with proto
$subref = sub (SIG) BLOCK;            # with signature
$subref = sub : ATTRS BLOCK;          # with attributes
$subref = sub (PROTO) : ATTRS BLOCK; # with proto and attributes
$subref = sub (SIG) : ATTRS BLOCK;   # with signature and attributes
```

To import subroutines:

```
use MODULE qw(NAME1 NAME2 NAME3);
```

To call subroutines:

```
NAME(LIST);    # & is optional with parentheses.
NAME LIST;     # Parentheses optional if predeclared/imported.
&NAME(LIST);   # Circumvent prototypes.
&NAME;         # Makes current @_ visible to called subroutine.
```

DESCRIPTION

Like many languages, Perl provides for user-defined subroutines. These may be located anywhere in the main program, loaded in from other files via the `do`, `require`, or `use` keywords, or generated on the fly using `eval` or anonymous subroutines. You can even call a function indirectly using a variable containing its name or a CODE reference.

The Perl model for function call and return values is simple: all functions are passed as parameters one single flat list of scalars, and all functions likewise return to their caller one single flat list of scalars. Any arrays or hashes in these call and return lists will collapse, losing their identities — but you may always use pass-by-reference instead to avoid this. Both call and return lists may contain as many or as few scalar elements as you'd like. (Often a function without an explicit return statement is called a subroutine, but there's really no difference from Perl's perspective.)

Any arguments passed in show up in the array `@_`. (They may also show up in lexical variables introduced by a signature; see "Signatures" below.) Therefore, if you called a function with two arguments, those would be stored in `$_[0]` and `$_[1]`. The array `@_` is a local array, but its elements are aliases for the actual scalar parameters. In particular, if an element `$_[0]` is updated, the corresponding argument is updated (or an error occurs if it is not updatable). If an argument is an array or hash element which did not exist when the function was called, that element is created only when (and if) it is modified or a reference to it is taken. (Some earlier versions of Perl created the element whether or not the element was assigned to.) Assigning to the whole array `@_` removes that aliasing, and does not update any arguments.

A `return` statement may be used to exit a subroutine, optionally specifying the returned value, which will be evaluated in the appropriate context (list, scalar, or void) depending on the context of the subroutine call. If you specify no return value, the subroutine returns an empty list in list context, the undefined value in scalar context, or nothing in void context. If you return one or more aggregates (arrays and hashes), these will be flattened together into one large indistinguishable list.

If no `return` is found and if the last statement is an expression, its value is returned. If the last statement is a loop control structure like a `foreach` or a `while`, the returned value is unspecified. The empty sub returns the empty list.

Aside from an experimental facility (see "Signatures" below), Perl does not have named formal parameters. In practice all you do is assign to a `my()` list of these. Variables that aren't declared to be private are global variables. For gory details on creating private variables, see "Private Variables via *my()*" and "Temporary Values via *local()*". To create protected environments for a set of functions in a separate package (and probably a separate file), see "Packages" in perlmod.

Example:

```
sub max {
    my $max = shift(@_);
    foreach $foo (@_) {
        $max = $foo if $max < $foo;
    }
    return $max;
}
$bestday = max($mon,$tue,$wed,$thu,$fri);
```

Example:

```
# get a line, combining continuation lines
#  that start with whitespace

sub get_line {
    $thisline = $lookahead;  # global variables!
    LINE: while (defined($lookahead = <STDIN>)) {
        if ($lookahead =~ /^[ \t]/) {
            $thisline .= $lookahead;
        }
        else {
            last LINE;
        }
    }
    return $thisline;
}

$lookahead = <STDIN>;       # get first line
while (defined($line = get_line())) {
    ...
}
```

Assigning to a list of private variables to name your arguments:

```
sub maybeset {
    my($key, $value) = @_;
    $Foo{$key} = $value unless $Foo{$key};
}
```

Because the assignment copies the values, this also has the effect of turning call-by-reference into call-by-value. Otherwise a function is free to do in-place modifications of `@_` and change its caller's values.

```
upcase_in($v1, $v2);  # this changes $v1 and $v2
sub upcase_in {
    for (@_) { tr/a-z/A-Z/ }
}
```

You aren't allowed to modify constants in this way, of course. If an argument were actually literal and you tried to change it, you'd take a (presumably fatal) exception. For example, this won't work:

```
upcase_in("frederick");
```

It would be much safer if the `upcase_in()` function were written to return a copy of its parameters instead of changing them in place:

```
($v3, $v4) = upcase($v1, $v2);  # this doesn't change $v1 and $v2
sub upcase {
    return unless defined wantarray;  # void context, do nothing
    my @parms = @_;
    for (@parms) { tr/a-z/A-Z/ }
    return wantarray ? @parms : $parms[0];
}
```

Notice how this (unprototyped) function doesn't care whether it was passed real scalars or arrays. Perl sees all arguments as one big, long, flat parameter list in `@_`. This is one area where Perl's simple argument-passing style shines. The `upcase()` function would work perfectly well without changing the `upcase()` definition even if we fed it things like this:

```
@newlist   = upcase(@list1, @list2);
@newlist   = upcase( split /:/, $var );
```

Do not, however, be tempted to do this:

```
(@a, @b)   = upcase(@list1, @list2);
```

Like the flattened incoming parameter list, the return list is also flattened on return. So all you have managed to do here is stored everything in `@a` and made `@b` empty. See "Pass by Reference" for alternatives.

A subroutine may be called using an explicit `&` prefix. The `&` is optional in modern Perl, as are parentheses if the subroutine has been predeclared. The `&` is *not* optional when just naming the subroutine, such as when it's used as an argument to *defined()* or *undef()*. Nor is it optional when you want to do an indirect subroutine call with a subroutine name or reference using the `&$subref()` or `&{$subref}()` constructs, although the `$subref->()` notation solves that problem. See perlref for more about all that.

Subroutines may be called recursively. If a subroutine is called using the `&` form, the argument list is optional, and if omitted, no `@_` array is set up for the subroutine: the `@_` array at the time of the call is visible to subroutine instead. This is an efficiency mechanism that new users may wish to avoid.

```
&foo(1,2,3);          # pass three arguments
foo(1,2,3);           # the same

foo();                # pass a null list
&foo();               # the same

&foo;                 # foo() get current args, like foo(@_) !!
foo;                  # like foo() IFF sub foo predeclared, else "foo"
```

Not only does the `&` form make the argument list optional, it also disables any prototype checking on arguments you do provide. This is partly for historical reasons, and partly for having a convenient way to cheat if you know what you're doing. See "Prototypes" below.

Since Perl 5.16.0, the `__SUB__` token is available under `use feature 'current_sub'` and `use 5.16.0`. It will evaluate to a reference to the currently-running sub, which allows for recursive calls without knowing your subroutine's name.

```
use 5.16.0;
my $factorial = sub {
  my ($x) = @_;
  return 1 if $x == 1;
  return($x * __SUB__->( $x - 1 ) );
};
```

The behavior of `__SUB__` within a regex code block (such as `/(?{...})/`) is subject to change.

Subroutines whose names are in all upper case are reserved to the Perl core, as are modules whose names are in all lower case. A subroutine in all capitals is a loosely-held convention meaning it will be called indirectly by the run-time system itself, usually due to a triggered event. Subroutines whose name start with a left parenthesis are also reserved the same way. The following is a list of some subroutines that currently do special, pre-defined things.

documented later in this document
AUTOLOAD

documented in perlmod
CLONE, CLONE_SKIP

documented in perlobj
DESTROY, DOES

documented in perltie
BINMODE, CLEAR, CLOSE, DELETE, DESTROY, EOF, EXISTS, EXTEND, FETCH, FETCHSIZE, FILENO, FIRSTKEY, GETC, NEXTKEY, OPEN, POP, PRINT, PRINTF, PUSH, READ, READLINE, SCALAR, SEEK, SHIFT, SPLICE, STORE, STORESIZE, TELL, TIEARRAY, TIEHANDLE, TIEHASH, TIESCALAR, UNSHIFT, UNTIE, WRITE

documented in PerlIO::via
BINMODE, CLEARERR, CLOSE, EOF, ERROR, FDOPEN, FILENO, FILL, FLUSH, OPEN, POPPED, PUSHED, READ, SEEK, SETLINEBUF, SYSOPEN, TELL, UNREAD, UTF8, WRITE

documented in perlfunc
import , unimport , INC

documented in UNIVERSAL
VERSION

documented in perldebguts
DB::DB, DB::sub, DB::lsub, DB::goto, DB::postponed

undocumented, used internally by the overload feature
any starting with (

The BEGIN, UNITCHECK, CHECK, INIT and END subroutines are not so much subroutines as named special code blocks, of which you can have more than one in a package, and which you can **not** call explicitly. See "BEGIN, UNITCHECK, CHECK, INIT and END" in perlmod

Signatures

WARNING: Subroutine signatures are experimental. The feature may be modified or removed in future versions of Perl.

Perl has an experimental facility to allow a subroutine's formal parameters to be introduced by special syntax, separate from the procedural code of the subroutine body. The formal parameter list is known as a *signature*. The facility must be enabled first by a pragmatic declaration, `use feature 'signatures'`, and it will produce a warning unless the "experimental::signatures" warnings category is disabled.

The signature is part of a subroutine's body. Normally the body of a subroutine is simply a braced block of code. When using a signature, the signature is a parenthesised list that goes immediately after the subroutine name (or, for anonymous subroutines, immediately after the `sub` keyword). The signature declares lexical variables that are in scope for the block. When the subroutine is called, the signature takes control first. It populates the signature variables from the list of arguments that were passed. If the argument list doesn't meet the requirements of the signature, then it will throw an exception. When the signature processing is complete, control passes to the block.

Positional parameters are handled by simply naming scalar variables in the signature. For example,

```
sub foo ($left, $right) {
    return $left + $right;
}
```

takes two positional parameters, which must be filled at runtime by two arguments. By default the parameters are mandatory, and it is not permitted to pass more arguments than expected. So the above is equivalent to

```
sub foo {
    die "Too many arguments for subroutine" unless @_ <= 2;
    die "Too few arguments for subroutine" unless @_ >= 2;
    my $left = $_[0];
    my $right = $_[1];
    return $left + $right;
}
```

An argument can be ignored by omitting the main part of the name from a parameter declaration, leaving just a bare `$` sigil. For example,

```
sub foo ($first, $, $third) {
    return "first=$first, third=$third";
}
```

Although the ignored argument doesn't go into a variable, it is still mandatory for the caller to pass it.

A positional parameter is made optional by giving a default value, separated from the parameter name by =:

```
sub foo ($left, $right = 0) {
    return $left + $right;
}
```

The above subroutine may be called with either one or two arguments. The default value expression is evaluated when the subroutine is called, so it may provide different default values for different calls. It is only evaluated if the argument was actually omitted from the call. For example,

```
my $auto_id = 0;
sub foo ($thing, $id = $auto_id++) {
    print "$thing has ID $id";
}
```

automatically assigns distinct sequential IDs to things for which no ID was supplied by the caller. A default value expression may also refer to parameters earlier in the signature, making the default for one parameter vary according to the earlier parameters. For example,

```
sub foo ($first_name, $surname, $nickname = $first_name) {
    print "$first_name $surname is known as \"$nickname\"";
}
```

An optional parameter can be nameless just like a mandatory parameter. For example,

```
sub foo ($thing, $ = 1) {
    print $thing;
}
```

The parameter's default value will still be evaluated if the corresponding argument isn't supplied, even though the value won't be stored anywhere. This is in case evaluating it has important side effects. However, it will be evaluated in void context, so if it doesn't have side effects and is not trivial it will generate a warning if the ''void'' warning category is enabled. If a nameless optional parameter's default value is not important, it may be omitted just as the parameter's name was:

```
sub foo ($thing, $=) {
    print $thing;
}
```

Optional positional parameters must come after all mandatory positional parameters. (If there are no mandatory positional parameters then an optional positional parameters can be the first thing in the signature.) If there are multiple optional positional parameters and not enough arguments are supplied to fill them all, they will be filled from left to right.

After positional parameters, additional arguments may be captured in a slurpy parameter. The simplest form of this is just an array variable:

```
sub foo ($filter, @inputs) {
    print $filter->($_) foreach @inputs;
}
```

With a slurpy parameter in the signature, there is no upper limit on how many arguments may be passed. A

slurpy array parameter may be nameless just like a positional parameter, in which case its only effect is to turn off the argument limit that would otherwise apply:

```
sub foo ($thing, @) {
    print $thing;
}
```

A slurpy parameter may instead be a hash, in which case the arguments available to it are interpreted as alternating keys and values. There must be as many keys as values: if there is an odd argument then an exception will be thrown. Keys will be stringified, and if there are duplicates then the later instance takes precedence over the earlier, as with standard hash construction.

```
sub foo ($filter, %inputs) {
    print $filter->($_, $inputs{$_}) foreach sort keys %inputs;
}
```

A slurpy hash parameter may be nameless just like other kinds of parameter. It still insists that the number of arguments available to it be even, even though they're not being put into a variable.

```
sub foo ($thing, %) {
    print $thing;
}
```

A slurpy parameter, either array or hash, must be the last thing in the signature. It may follow mandatory and optional positional parameters; it may also be the only thing in the signature. Slurpy parameters cannot have default values: if no arguments are supplied for them then you get an empty array or empty hash.

A signature may be entirely empty, in which case all it does is check that the caller passed no arguments:

```
sub foo () {
    return 123;
}
```

When using a signature, the arguments are still available in the special array variable `@_`, in addition to the lexical variables of the signature. There is a difference between the two ways of accessing the arguments: `@_` *aliases* the arguments, but the signature variables get *copies* of the arguments. So writing to a signature variable only changes that variable, and has no effect on the caller's variables, but writing to an element of `@_` modifies whatever the caller used to supply that argument.

There is a potential syntactic ambiguity between signatures and prototypes (see "Prototypes"), because both start with an opening parenthesis and both can appear in some of the same places, such as just after the name in a subroutine declaration. For historical reasons, when signatures are not enabled, any opening parenthesis in such a context will trigger very forgiving prototype parsing. Most signatures will be interpreted as prototypes in those circumstances, but won't be valid prototypes. (A valid prototype cannot contain any alphabetic character.) This will lead to somewhat confusing error messages.

To avoid ambiguity, when signatures are enabled the special syntax for prototypes is disabled. There is no attempt to guess whether a parenthesised group was intended to be a prototype or a signature. To give a subroutine a prototype under these circumstances, use a prototype attribute. For example,

```
sub foo :prototype($) { $_[0] }
```

It is entirely possible for a subroutine to have both a prototype and a signature. They do different jobs: the prototype affects compilation of calls to the subroutine, and the signature puts argument values into lexical variables at runtime. You can therefore write

```
sub foo ($left, $right) : prototype($$) {
    return $left + $right;
}
```

The prototype attribute, and any other attributes, come after the signature.

Private Variables via *my()*

Synopsis:

```
    my $foo;            # declare $foo lexically local
    my (@wid, %get);    # declare list of variables local
    my $foo = "flurp";  # declare $foo lexical, and init it
    my @oof = @bar;     # declare @oof lexical, and init it
    my $x : Foo = $y;   # similar, with an attribute applied
```

WARNING: The use of attribute lists on `my` declarations is still evolving. The current semantics and interface are subject to change. See attributes and Attribute::Handlers.

The `my` operator declares the listed variables to be lexically confined to the enclosing block, conditional (`if`/`unless`/`elsif`/`else`), loop (`for`/`foreach`/`while`/`until`/`continue`), subroutine, `eval`, or `do`/`require`/`use`'d file. If more than one value is listed, the list must be placed in parentheses. All listed elements must be legal lvalues. Only alphanumeric identifiers may be lexically scoped — magical built-ins like `$/` must currently be `local`ized with `local` instead.

Unlike dynamic variables created by the `local` operator, lexical variables declared with `my` are totally hidden from the outside world, including any called subroutines. This is true if it's the same subroutine called from itself or elsewhere — every call gets its own copy.

This doesn't mean that a `my` variable declared in a statically enclosing lexical scope would be invisible. Only dynamic scopes are cut off. For example, the `bumpx()` function below has access to the lexical `$x` variable because both the `my` and the `sub` occurred at the same scope, presumably file scope.

```
    my $x = 10;
    sub bumpx { $x++ }
```

An `eval()`, however, can see lexical variables of the scope it is being evaluated in, so long as the names aren't hidden by declarations within the `eval()` itself. See perlref.

The parameter list to *my()* may be assigned to if desired, which allows you to initialize your variables. (If no initializer is given for a particular variable, it is created with the undefined value.) Commonly this is used to name input parameters to a subroutine. Examples:

```
    $arg = "fred";        # "global" variable
    $n = cube_root(27);
    print "$arg thinks the root is $n\n";
 fred thinks the root is 3

    sub cube_root {
        my $arg = shift;  # name doesn't matter
        $arg **= 1/3;
        return $arg;
    }
```

The `my` is simply a modifier on something you might assign to. So when you do assign to variables in its argument list, `my` doesn't change whether those variables are viewed as a scalar or an array. So

```
    my ($foo) = <STDIN>;                # WRONG?
    my @FOO = <STDIN>;
```

both supply a list context to the right-hand side, while

```
    my $foo = <STDIN>;
```

supplies a scalar context. But the following declares only one variable:

```
    my $foo, $bar = 1;                  # WRONG
```

That has the same effect as

```
    my $foo;
    $bar = 1;
```

The declared variable is not introduced (is not visible) until after the current statement. Thus,

```
    my $x = $x;
```

can be used to initialize a new `$x` with the value of the old `$x`, and the expression

```
    my $x = 123 and $x == 123
```

is false unless the old `$x` happened to have the value `123`.

Lexical scopes of control structures are not bounded precisely by the braces that delimit their controlled blocks; control expressions are part of that scope, too. Thus in the loop

```
while (my $line = <>) {
    $line = lc $line;
} continue {
    print $line;
}
```

the scope of `$line` extends from its declaration throughout the rest of the loop construct (including the `continue` clause), but not beyond it. Similarly, in the conditional

```
if ((my $answer = <STDIN>) =~ /^yes$/i) {
    user_agrees();
} elsif ($answer =~ /^no$/i) {
    user_disagrees();
} else {
    chomp $answer;
    die "'$answer' is neither 'yes' nor 'no'";
}
```

the scope of `$answer` extends from its declaration through the rest of that conditional, including any `elsif` and `else` clauses, but not beyond it. See "Simple Statements" in perlsyn for information on the scope of variables in statements with modifiers.

The `foreach` loop defaults to scoping its index variable dynamically in the manner of `local`. However, if the index variable is prefixed with the keyword `my`, or if there is already a lexical by that name in scope, then a new lexical is created instead. Thus in the loop

```
for my $i (1, 2, 3) {
    some_function();
}
```

the scope of `$i` extends to the end of the loop, but not beyond it, rendering the value of `$i` inaccessible within `some_function()`.

Some users may wish to encourage the use of lexically scoped variables. As an aid to catching implicit uses to package variables, which are always global, if you say

```
use strict 'vars';
```

then any variable mentioned from there to the end of the enclosing block must either refer to a lexical variable, be predeclared via `our` or `use vars`, or else must be fully qualified with the package name. A compilation error results otherwise. An inner block may countermand this with `no strict 'vars'`.

A `my` has both a compile-time and a run-time effect. At compile time, the compiler takes notice of it. The principal usefulness of this is to quiet `use strict 'vars'`, but it is also essential for generation of closures as detailed in perlref. Actual initialization is delayed until run time, though, so it gets executed at the appropriate time, such as each time through a loop, for example.

Variables declared with `my` are not part of any package and are therefore never fully qualified with the package name. In particular, you're not allowed to try to make a package variable (or other global) lexical:

```
my $pack::var;      # ERROR!  Illegal syntax
```

In fact, a dynamic variable (also known as package or global variables) are still accessible using the fully qualified `::` notation even while a lexical of the same name is also visible:

```
package main;
local $x = 10;
my    $x = 20;
print "$x and $::x\n";
```

That will print out `20` and `10`.

You may declare `my` variables at the outermost scope of a file to hide any such identifiers from the world outside that file. This is similar in spirit to C's static variables when they are used at the file level. To do

this with a subroutine requires the use of a closure (an anonymous function that accesses enclosing lexicals). If you want to create a private subroutine that cannot be called from outside that block, it can declare a lexical variable containing an anonymous sub reference:

```
my $secret_version = '1.001-beta';
my $secret_sub = sub { print $secret_version };
&$secret_sub();
```

As long as the reference is never returned by any function within the module, no outside module can see the subroutine, because its name is not in any package's symbol table. Remember that it's not *REALLY* called `$some_pack::secret_version` or anything; it's just `$secret_version`, unqualified and unqualifiable.

This does not work with object methods, however; all object methods have to be in the symbol table of some package to be found. See "Function Templates" in perlref for something of a work-around to this.

Persistent Private Variables

There are two ways to build persistent private variables in Perl 5.10. First, you can simply use the `state` feature. Or, you can use closures, if you want to stay compatible with releases older than 5.10.

Persistent variables via state()

Beginning with Perl 5.10.0, you can declare variables with the `state` keyword in place of `my`. For that to work, though, you must have enabled that feature beforehand, either by using the `feature` pragma, or by using `-E` on one-liners (see feature). Beginning with Perl 5.16, the `CORE::state` form does not require the `feature` pragma.

The `state` keyword creates a lexical variable (following the same scoping rules as `my`) that persists from one subroutine call to the next. If a state variable resides inside an anonymous subroutine, then each copy of the subroutine has its own copy of the state variable. However, the value of the state variable will still persist between calls to the same copy of the anonymous subroutine. (Don't forget that `sub { ... }` creates a new subroutine each time it is executed.)

For example, the following code maintains a private counter, incremented each time the *gimme_another()* function is called:

```
use feature 'state';
sub gimme_another { state $x; return ++$x }
```

And this example uses anonymous subroutines to create separate counters:

```
use feature 'state';
sub create_counter {
    return sub { state $x; return ++$x }
}
```

Also, since `$x` is lexical, it can't be reached or modified by any Perl code outside.

When combined with variable declaration, simple scalar assignment to `state` variables (as in `state $x = 42`) is executed only the first time. When such statements are evaluated subsequent times, the assignment is ignored. The behavior of this sort of assignment to non-scalar variables is undefined.

Persistent variables with closures

Just because a lexical variable is lexically (also called statically) scoped to its enclosing block, `eval`, or `do` FILE, this doesn't mean that within a function it works like a C static. It normally works more like a C auto, but with implicit garbage collection.

Unlike local variables in C or C++, Perl's lexical variables don't necessarily get recycled just because their scope has exited. If something more permanent is still aware of the lexical, it will stick around. So long as something else references a lexical, that lexical won't be freed — which is as it should be. You wouldn't want memory being free until you were done using it, or kept around once you were done. Automatic garbage collection takes care of this for you.

This means that you can pass back or save away references to lexical variables, whereas to return a pointer to a C auto is a grave error. It also gives us a way to simulate C's function statics. Here's a mechanism for giving a function private variables with both lexical scoping and a static lifetime. If you do want to create something like C's static variables, just enclose the whole function in an extra block, and put the static variable outside the function but in the block.

```
{
    my $secret_val = 0;
    sub gimme_another {
        return ++$secret_val;
    }
}
# $secret_val now becomes unreachable by the outside
# world, but retains its value between calls to gimme_another
```

If this function is being sourced in from a separate file via `require` or `use`, then this is probably just fine. If it's all in the main program, you'll need to arrange for the `my` to be executed early, either by putting the whole block above your main program, or more likely, placing merely a `BEGIN` code block around it to make sure it gets executed before your program starts to run:

```
BEGIN {
    my $secret_val = 0;
    sub gimme_another {
        return ++$secret_val;
    }
}
```

See "BEGIN, UNITCHECK, CHECK, INIT and END" in perlmod about the special triggered code blocks, `BEGIN`, `UNITCHECK`, `CHECK`, `INIT` and `END`.

If declared at the outermost scope (the file scope), then lexicals work somewhat like C's file statics. They are available to all functions in that same file declared below them, but are inaccessible from outside that file. This strategy is sometimes used in modules to create private variables that the whole module can see.

Temporary Values via *local()*

WARNING: In general, you should be using `my` instead of `local`, because it's faster and safer. Exceptions to this include the global punctuation variables, global filehandles and formats, and direct manipulation of the Perl symbol table itself. `local` is mostly used when the current value of a variable must be visible to called subroutines.

Synopsis:

```
# localization of values

local $foo;                 # make $foo dynamically local
local (@wid, %get);         # make list of variables local
local $foo = "flurp";       # make $foo dynamic, and init it
local @oof = @bar;          # make @oof dynamic, and init it

local $hash{key} = "val";   # sets a local value for this hash entry
delete local $hash{key};    # delete this entry for the current block
local ($cond ? $v1 : $v2);  # several types of lvalues support
                            # localization

# localization of symbols

local *FH;                  # localize $FH, @FH, %FH, &FH  ...
local *merlyn = *randal;    # now $merlyn is really $randal, plus
                            #     @merlyn is really @randal, etc
local *merlyn = 'randal';   # SAME THING: promote 'randal' to *randal
local *merlyn = \$randal;   # just alias $merlyn, not @merlyn etc
```

A `local` modifies its listed variables to be "local" to the enclosing block, `eval`, or `do FILE`--and to *any subroutine called from within that block*. A `local` just gives temporary values to global (meaning package) variables. It does *not* create a local variable. This is known as dynamic scoping. Lexical scoping is done with `my`, which works more like C's auto declarations.

Some types of lvalues can be localized as well: hash and array elements and slices, conditionals (provided that their result is always localizable), and symbolic references. As for simple variables, this creates new, dynamically scoped values.

If more than one variable or expression is given to `local`, they must be placed in parentheses. This operator works by saving the current values of those variables in its argument list on a hidden stack and restoring them upon exiting the block, subroutine, or eval. This means that called subroutines can also reference the local variable, but not the global one. The argument list may be assigned to if desired, which allows you to initialize your local variables. (If no initializer is given for a particular variable, it is created with an undefined value.)

Because `local` is a run-time operator, it gets executed each time through a loop. Consequently, it's more efficient to localize your variables outside the loop.

Grammatical note on local()

A `local` is simply a modifier on an lvalue expression. When you assign to a `local`ized variable, the `local` doesn't change whether its list is viewed as a scalar or an array. So

```
local($foo) = <STDIN>;
local @FOO = <STDIN>;
```

both supply a list context to the right-hand side, while

```
local $foo = <STDIN>;
```

supplies a scalar context.

Localization of special variables

If you localize a special variable, you'll be giving a new value to it, but its magic won't go away. That means that all side-effects related to this magic still work with the localized value.

This feature allows code like this to work :

```
# Read the whole contents of FILE in $slurp
{ local $/ = undef; $slurp = <FILE>; }
```

Note, however, that this restricts localization of some values ; for example, the following statement dies, as of perl 5.10.0, with an error *Modification of a read-only value attempted*, because the `$1` variable is magical and read-only :

```
local $1 = 2;
```

One exception is the default scalar variable: starting with perl 5.14 `local($_)` will always strip all magic from `$_`, to make it possible to safely reuse `$_` in a subroutine.

WARNING: Localization of tied arrays and hashes does not currently work as described. This will be fixed in a future release of Perl; in the meantime, avoid code that relies on any particular behavior of localising tied arrays or hashes (localising individual elements is still okay). See ''Localising Tied Arrays and Hashes Is Broken'' in perl58delta for more details.

Localization of globs

The construct

```
local *name;
```

creates a whole new symbol table entry for the glob `name` in the current package. That means that all variables in its glob slot ($name, `@name`, `%name`, &name, and the `name` filehandle) are dynamically reset.

This implies, among other things, that any magic eventually carried by those variables is locally lost. In other words, saying `local */` will not have any effect on the internal value of the input record separator.

Localization of elements of composite types

It's also worth taking a moment to explain what happens when you `local`ize a member of a composite type (i.e. an array or hash element). In this case, the element is `local`ized *by name*. This means that when the scope of the `local()` ends, the saved value will be restored to the hash element whose key was named in the `local()`, or the array element whose index was named in the `local()`. If that element was deleted while the `local()` was in effect (e.g. by a `delete()` from a hash or a `shift()` of an array), it will spring back into existence, possibly extending an array and filling in the skipped elements with `undef`. For instance, if you say

```
%hash = ( 'This' => 'is', 'a' => 'test' );
@ary  = ( 0..5 );
{
     local($ary[5]) = 6;
     local($hash{'a'}) = 'drill';
     while (my $e = pop(@ary)) {
         print "$e . . .\n";
         last unless $e > 3;
     }
     if (@ary) {
         $hash{'only a'} = 'test';
         delete $hash{'a'};
     }
}
print join(' ', map { "$_ $hash{$_}" } sort keys %hash),".\n";
print "The array has ",scalar(@ary)," elements: ",
      join(', ', map { defined $_ ? $_ : 'undef' } @ary),"\n";
```

Perl will print

```
6 . . .
4 . . .
3 . . .
This is a test only a test.
The array has 6 elements: 0, 1, 2, undef, undef, 5
```

The behavior of *local()* on non-existent members of composite types is subject to change in future.

Localized deletion of elements of composite types

You can use the `delete local $array[$idx]` and `delete local $hash{key}` constructs to delete a composite type entry for the current block and restore it when it ends. They return the array/hash value before the localization, which means that they are respectively equivalent to

```
do {
    my $val = $array[$idx];
    local  $array[$idx];
    delete $array[$idx];
    $val
}
```

and

```
do {
    my $val = $hash{key};
    local  $hash{key};
    delete $hash{key};
    $val
}
```

except that for those the `local` is scoped to the `do` block. Slices are also accepted.

```
my %hash = (
 a => [ 7, 8, 9 ],
 b => 1,
)

{
 my $a = delete local $hash{a};
 # $a is [ 7, 8, 9 ]
 # %hash is (b => 1)

 {
  my @nums = delete local @$a[0, 2]
  # @nums is (7, 9)
```

```
  # $a is [ undef, 8 ]

  $a[0] = 999; # will be erased when the scope ends
 }
 # $a is back to [ 7, 8, 9 ]

}
# %hash is back to its original state
```

Lvalue subroutines

It is possible to return a modifiable value from a subroutine. To do this, you have to declare the subroutine to return an lvalue.

```
my $val;
sub canmod : lvalue {
    $val;  # or:  return $val;
}
sub nomod {
    $val;
}

canmod() = 5;   # assigns to $val
nomod()  = 5;   # ERROR
```

The scalar/list context for the subroutine and for the right-hand side of assignment is determined as if the subroutine call is replaced by a scalar. For example, consider:

```
data(2,3) = get_data(3,4);
```

Both subroutines here are called in a scalar context, while in:

```
(data(2,3)) = get_data(3,4);
```

and in:

```
(data(2),data(3)) = get_data(3,4);
```

all the subroutines are called in a list context.

Lvalue subroutines are convenient, but you have to keep in mind that, when used with objects, they may violate encapsulation. A normal mutator can check the supplied argument before setting the attribute it is protecting, an lvalue subroutine cannot. If you require any special processing when storing and retrieving the values, consider using the CPAN module Sentinel or something similar.

Lexical Subroutines

Beginning with Perl 5.18, you can declare a private subroutine with `my` or `state`. As with state variables, the `state` keyword is only available under `use feature 'state'` or `use 5.010` or higher.

Prior to Perl 5.26, lexical subroutines were deemed experimental and were available only under the `use feature 'lexical_subs'` pragma. They also produced a warning unless the "experimental::lexical_subs" warnings category was disabled.

These subroutines are only visible within the block in which they are declared, and only after that declaration:

```
# Include these two lines if your code is intended to run under Perl
# versions earlier than 5.26.
no warnings "experimental::lexical_subs";
use feature 'lexical_subs';

foo();              # calls the package/global subroutine
state sub foo {
    foo();          # also calls the package subroutine
}
foo();              # calls "state" sub
my $ref = \&foo;    # take a reference to "state" sub
```

```
my sub bar { ... }
bar();              # calls "my" sub
```

To use a lexical subroutine from inside the subroutine itself, you must predeclare it. The `sub foo {...}` subroutine definition syntax respects any previous `my sub;` or `state sub;` declaration.

```
my sub baz;         # predeclaration
sub baz {           # define the "my" sub
    baz();          # recursive call
}
```

`state sub` vs `my sub`

What is the difference between "state" subs and "my" subs? Each time that execution enters a block when "my" subs are declared, a new copy of each sub is created. "State" subroutines persist from one execution of the containing block to the next.

So, in general, "state" subroutines are faster. But "my" subs are necessary if you want to create closures:

```
sub whatever {
    my $x = shift;
    my sub inner {
        ... do something with $x ...
    }
    inner();
}
```

In this example, a new `$x` is created when `whatever` is called, and also a new `inner`, which can see the new `$x`. A "state" sub will only see the `$x` from the first call to `whatever`.

`our` subroutines

Like `our $variable`, `our sub` creates a lexical alias to the package subroutine of the same name.

The two main uses for this are to switch back to using the package sub inside an inner scope:

```
sub foo { ... }

sub bar {
    my sub foo { ... }
    {
        # need to use the outer foo here
        our sub foo;
        foo();
    }
}
```

and to make a subroutine visible to other packages in the same scope:

```
package MySneakyModule;

our sub do_something { ... }

sub do_something_with_caller {
    package DB;
    () = caller 1;          # sets @DB::args
    do_something(@args);    # uses MySneakyModule::do_something
}
```

Passing Symbol Table Entries (typeglobs)

WARNING: The mechanism described in this section was originally the only way to simulate pass-by-reference in older versions of Perl. While it still works fine in modern versions, the new reference mechanism is generally easier to work with. See below.

Sometimes you don't want to pass the value of an array to a subroutine but rather the name of it, so that the subroutine can modify the global copy of it rather than working with a local copy. In perl you can refer to all objects of a particular name by prefixing the name with a star: `*foo`. This is often known as a "typeglob", because the star on the front can be thought of as a wildcard match for all the funny prefix

characters on variables and subroutines and such.

When evaluated, the typeglob produces a scalar value that represents all the objects of that name, including any filehandle, format, or subroutine. When assigned to, it causes the name mentioned to refer to whatever `*` value was assigned to it. Example:

```
sub doubleary {
    local(*someary) = @_;
    foreach $elem (@someary) {
        $elem *= 2;
    }
}
doubleary(*foo);
doubleary(*bar);
```

Scalars are already passed by reference, so you can modify scalar arguments without using this mechanism by referring explicitly to `$_[0]` etc. You can modify all the elements of an array by passing all the elements as scalars, but you have to use the `*` mechanism (or the equivalent reference mechanism) to `push`, `pop`, or change the size of an array. It will certainly be faster to pass the typeglob (or reference).

Even if you don't want to modify an array, this mechanism is useful for passing multiple arrays in a single LIST, because normally the LIST mechanism will merge all the array values so that you can't extract out the individual arrays. For more on typeglobs, see "Typeglobs and Filehandles" in perldata.

When to Still Use *local()*

Despite the existence of `my`, there are still three places where the `local` operator still shines. In fact, in these three places, you *must* use `local` instead of `my`.

1. You need to give a global variable a temporary value, especially `$_`.

   The global variables, like `@ARGV` or the punctuation variables, must be `local`ized with `local()`. This block reads in */etc/motd*, and splits it up into chunks separated by lines of equal signs, which are placed in `@Fields`.

   ```
   {
       local @ARGV = ("/etc/motd");
       local $/ = undef;
       local $_ = <>;
       @Fields = split /^\s*=+\s*$/;
   }
   ```

   It particular, it's important to `local`ize `$_` in any routine that assigns to it. Look out for implicit assignments in `while` conditionals.

2. You need to create a local file or directory handle or a local function.

   A function that needs a filehandle of its own must use `local()` on a complete typeglob. This can be used to create new symbol table entries:

   ```
   sub ioqueue {
       local  (*READER, *WRITER);    # not my!
       pipe    (READER,  WRITER)     or die "pipe: $!";
       return (*READER, *WRITER);
   }
   ($head, $tail) = ioqueue();
   ```

   See the Symbol module for a way to create anonymous symbol table entries.

   Because assignment of a reference to a typeglob creates an alias, this can be used to create what is effectively a local function, or at least, a local alias.

```
{
    local *grow = \&shrink; # only until this block exits
    grow();                 # really calls shrink()
    move();                 # if move() grow()s, it shrink()s too
}
grow();                     # get the real grow() again
```

See "Function Templates" in perlref for more about manipulating functions by name in this way.

3. You want to temporarily change just one element of an array or hash.

   You can `local`ize just one element of an aggregate. Usually this is done on dynamics:

```
{
    local $SIG{INT} = 'IGNORE';
    funct();                            # uninterruptible
}
# interruptibility automatically restored here
```

   But it also works on lexically declared aggregates.

Pass by Reference

If you want to pass more than one array or hash into a function — or return them from it — and have them maintain their integrity, then you're going to have to use an explicit pass-by-reference. Before you do that, you need to understand references as detailed in perlref. This section may not make much sense to you otherwise.

Here are a few simple examples. First, let's pass in several arrays to a function and have it `pop` all of then, returning a new list of all their former last elements:

```
@tailings = popmany ( \@a, \@b, \@c, \@d );

sub popmany {
    my $aref;
    my @retlist;
    foreach $aref ( @_ ) {
        push @retlist, pop @$aref;
    }
    return @retlist;
}
```

Here's how you might write a function that returns a list of keys occurring in all the hashes passed to it:

```
@common = inter( \%foo, \%bar, \%joe );
sub inter {
    my ($k, $href, %seen); # locals
    foreach $href (@_) {
        while ( $k = each %$href ) {
            $seen{$k}++;
        }
    }
    return grep { $seen{$_} == @_ } keys %seen;
}
```

So far, we're using just the normal list return mechanism. What happens if you want to pass or return a hash? Well, if you're using only one of them, or you don't mind them concatenating, then the normal calling convention is ok, although a little expensive.

Where people get into trouble is here:

```
    (@a, @b) = func(@c, @d);
or
    (%a, %b) = func(%c, %d);
```

That syntax simply won't work. It sets just `@a` or `%a` and clears the `@b` or `%b`. Plus the function didn't get passed into two separate arrays or hashes: it got one long list in `@_`, as always.

If you can arrange for everyone to deal with this through references, it's cleaner code, although not so nice to look at. Here's a function that takes two array references as arguments, returning the two array elements in order of how many elements they have in them:

```
($aref, $bref) = func(\@c, \@d);
print "@$aref has more than @$bref\n";
sub func {
    my ($cref, $dref) = @_;
    if (@$cref > @$dref) {
        return ($cref, $dref);
    } else {
        return ($dref, $cref);
    }
}
```

It turns out that you can actually do this also:

```
(*a, *b) = func(\@c, \@d);
print "@a has more than @b\n";
sub func {
    local (*c, *d) = @_;
    if (@c > @d) {
        return (\@c, \@d);
    } else {
        return (\@d, \@c);
    }
}
```

Here we're using the typeglobs to do symbol table aliasing. It's a tad subtle, though, and also won't work if you're using `my` variables, because only globals (even in disguise as `locals`) are in the symbol table.

If you're passing around filehandles, you could usually just use the bare typeglob, like `*STDOUT`, but typeglobs references work, too. For example:

```
splutter(\*STDOUT);
sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}

$rec = get_rec(\*STDIN);
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

If you're planning on generating new filehandles, you could do this. Notice to pass back just the bare *FH, not its reference.

```
sub openit {
    my $path = shift;
    local *FH;
    return open (FH, $path) ? *FH : undef;
}
```

Prototypes

Perl supports a very limited kind of compile-time argument checking using function prototyping. This can be declared in either the PROTO section or with a prototype attribute. If you declare either of

```
sub mypush (\@@)
sub mypush :prototype(\@@)
```

then `mypush()` takes arguments exactly like `push()` does.

If subroutine signatures are enabled (see "Signatures"), then the shorter PROTO syntax is unavailable, because it would clash with signatures. In that case, a prototype can only be declared in the form of an

attribute.

The function declaration must be visible at compile time. The prototype affects only interpretation of new-style calls to the function, where new-style is defined as not using the `&` character. In other words, if you call it like a built-in function, then it behaves like a built-in function. If you call it like an old-fashioned subroutine, then it behaves like an old-fashioned subroutine. It naturally falls out from this rule that prototypes have no influence on subroutine references like `\&foo` or on indirect subroutine calls like `&{$subref}` or `$subref->()`.

Method calls are not influenced by prototypes either, because the function to be called is indeterminate at compile time, since the exact code called depends on inheritance.

Because the intent of this feature is primarily to let you define subroutines that work like built-in functions, here are prototypes for some other functions that parse almost exactly like the corresponding built-in.

```
Declared as             Called as

sub mylink ($$)         mylink $old, $new
sub myvec ($$$)         myvec $var, $offset, 1
sub myindex ($$;$)      myindex &getstring, "substr"
sub mysyswrite ($$$;$)  mysyswrite $buf, 0, length($buf) - $off, $off
sub myreverse (@)       myreverse $a, $b, $c
sub myjoin ($@)         myjoin ":", $a, $b, $c
sub mypop (\@)          mypop @array
sub mysplice (\@$$@)    mysplice @array, 0, 2, @pushme
sub mykeys (\[%@])      mykeys %{$hashref}
sub myopen (*;$)        myopen HANDLE, $name
sub mypipe (**)         mypipe READHANDLE, WRITEHANDLE
sub mygrep (&@)         mygrep { /foo/ } $a, $b, $c
sub myrand (;$)         myrand 42
sub mytime ()           mytime
```

Any backslashed prototype character represents an actual argument that must start with that character (optionally preceded by `my`, `our` or `local`), with the exception of `$`, which will accept any scalar lvalue expression, such as `$foo = 7` or `my_function()->[0]`. The value passed as part of `@_` will be a reference to the actual argument given in the subroutine call, obtained by applying `\` to that argument.

You can use the `\[]` backslash group notation to specify more than one allowed argument type. For example:

```
sub myref (\[$@%&*])
```

will allow calling *myref()* as

```
myref $var
myref @array
myref %hash
myref &sub
myref *glob
```

and the first argument of *myref()* will be a reference to a scalar, an array, a hash, a code, or a glob.

Unbackslashed prototype characters have special meanings. Any unbackslashed `@` or `%` eats all remaining arguments, and forces list context. An argument represented by `$` forces scalar context. An `&` requires an anonymous subroutine, which, if passed as the first argument, does not require the `sub` keyword or a subsequent comma.

A `*` allows the subroutine to accept a bareword, constant, scalar expression, typeglob, or a reference to a typeglob in that slot. The value will be available to the subroutine either as a simple scalar, or (in the latter two cases) as a reference to the typeglob. If you wish to always convert such arguments to a typeglob reference, use *Symbol::qualify_to_ref()* as follows:

```
use Symbol 'qualify_to_ref';

sub foo (*) {
    my $fh = qualify_to_ref(shift, caller);
```

```
    ...
}
```

The + prototype is a special alternative to `$` that will act like `\[@%]` when given a literal array or hash variable, but will otherwise force scalar context on the argument. This is useful for functions which should accept either a literal array or an array reference as the argument:

```
sub mypush (+@) {
    my $aref = shift;
    die "Not an array or arrayref" unless ref $aref eq 'ARRAY';
    push @$aref, @_;
}
```

When using the + prototype, your function must check that the argument is of an acceptable type.

A semicolon (`;`) separates mandatory arguments from optional arguments. It is redundant before `@` or `%`, which gobble up everything else.

As the last character of a prototype, or just before a semicolon, a `@` or a `%`, you can use _ in place of `$`: if this argument is not provided, `$_` will be used instead.

Note how the last three examples in the table above are treated specially by the parser. `mygrep()` is parsed as a true list operator, `myrand()` is parsed as a true unary operator with unary precedence the same as `rand()`, and `mytime()` is truly without arguments, just like `time()`. That is, if you say

```
mytime +2;
```

you'll get `mytime() + 2`, not `mytime(2)`, which is how it would be parsed without a prototype. If you want to force a unary function to have the same precedence as a list operator, add `;` to the end of the prototype:

```
sub mygetprotobynumber($;);
mygetprotobynumber $a > $b; # parsed as mygetprotobynumber($a > $b)
```

The interesting thing about `&` is that you can generate new syntax with it, provided it's in the initial position:

```
sub try (&@) {
    my($try,$catch) = @_;
    eval { &$try };
    if ($@) {
        local $_ = $@;
        &$catch;
    }
}
sub catch (&) { $_[0] }

try {
    die "phooey";
} catch {
    /phooey/ and print "unphooey\n";
};
```

That prints `"unphooey"`. (Yes, there are still unresolved issues having to do with visibility of `@_`. I'm ignoring that question for the moment. (But note that if we make `@_` lexically scoped, those anonymous subroutines can act like closures... (Gee, is this sounding a little Lispish? (Never mind.))))

And here's a reimplementation of the Perl `grep` operator:

```
sub mygrep (&@) {
    my $code = shift;
    my @result;
    foreach $_ (@_) {
        push(@result, $_) if &$code;
    }
    @result;
}
```

Some folks would prefer full alphanumeric prototypes. Alphanumerics have been intentionally left out of prototypes for the express purpose of someday in the future adding named, formal parameters. The current mechanism's main goal is to let module writers provide better diagnostics for module users. Larry feels the notation quite understandable to Perl programmers, and that it will not intrude greatly upon the meat of the module, nor make it harder to read. The line noise is visually encapsulated into a small pill that's easy to swallow.

If you try to use an alphanumeric sequence in a prototype you will generate an optional warning – "Illegal character in prototype...". Unfortunately earlier versions of Perl allowed the prototype to be used as long as its prefix was a valid prototype. The warning may be upgraded to a fatal error in a future version of Perl once the majority of offending code is fixed.

It's probably best to prototype new functions, not retrofit prototyping into older ones. That's because you must be especially careful about silent impositions of differing list versus scalar contexts. For example, if you decide that a function should take just one parameter, like this:

```
sub func ($) {
    my $n = shift;
    print "you gave me $n\n";
}
```

and someone has been calling it with an array or expression returning a list:

```
func(@foo);
func( $text =~ /\w+/g );
```

Then you've just supplied an automatic `scalar` in front of their argument, which can be more than a bit surprising. The old `@foo` which used to hold one thing doesn't get passed in. Instead, `func()` now gets passed in a `1`; that is, the number of elements in `@foo`. And the `m//g` gets called in scalar context so instead of a list of words it returns a boolean result and advances `pos($text)`. Ouch!

If a sub has both a PROTO and a BLOCK, the prototype is not applied until after the BLOCK is completely defined. This means that a recursive function with a prototype has to be predeclared for the prototype to take effect, like so:

```
sub foo($$);
sub foo($$) {
        foo 1, 2;
}
```

This is all very powerful, of course, and should be used only in moderation to make the world a better place.

Constant Functions

Functions with a prototype of `()` are potential candidates for inlining. If the result after optimization and constant folding is either a constant or a lexically-scoped scalar which has no other references, then it will be used in place of function calls made without `&`. Calls made using `&` are never inlined. (See *constant.pm* for an easy way to declare most constants.)

The following functions would all be inlined:

```
sub pi ()           { 3.14159 }             # Not exact, but close.
sub PI ()           { 4 * atan2 1, 1 }      # As good as it gets,
                                            # and it's inlined, too!
sub ST_DEV ()       { 0 }
sub ST_INO ()       { 1 }
```

```
sub FLAG_FOO ()     { 1 << 8 }
sub FLAG_BAR ()     { 1 << 9 }
sub FLAG_MASK ()    { FLAG_FOO | FLAG_BAR }

sub OPT_BAZ ()      { not (0x1B58 & FLAG_MASK) }

sub N () { int(OPT_BAZ) / 3 }

sub FOO_SET () { 1 if FLAG_MASK & FLAG_FOO }
sub FOO_SET2 () { if (FLAG_MASK & FLAG_FOO) { 1 } }
```

(Be aware that the last example was not always inlined in Perl 5.20 and earlier, which did not behave consistently with subroutines containing inner scopes.) You can countermand inlining by using an explicit `return`:

```
sub baz_val () {
    if (OPT_BAZ) {
        return 23;
    }
    else {
        return 42;
    }
}
sub bonk_val () { return 12345 }
```

As alluded to earlier you can also declare inlined subs dynamically at BEGIN time if their body consists of a lexically-scoped scalar which has no other references. Only the first example here will be inlined:

```
BEGIN {
    my $var = 1;
    no strict 'refs';
    *INLINED = sub () { $var };
}

BEGIN {
    my $var = 1;
    my $ref = \$var;
    no strict 'refs';
    *NOT_INLINED = sub () { $var };
}
```

A not so obvious caveat with this (see [RT #79908]) is that the variable will be immediately inlined, and will stop behaving like a normal lexical variable, e.g. this will print `79907`, not `79908`:

```
BEGIN {
    my $x = 79907;
    *RT_79908 = sub () { $x };
    $x++;
}
print RT_79908(); # prints 79907
```

As of Perl 5.22, this buggy behavior, while preserved for backward compatibility, is detected and emits a deprecation warning. If you want the subroutine to be inlined (with no warning), make sure the variable is not used in a context where it could be modified aside from where it is declared.

```
# Fine, no warning
BEGIN {
    my $x = 54321;
    *INLINED = sub () { $x };
}
# Warns.  Future Perl versions will stop inlining it.
BEGIN {
    my $x;
    $x = 54321;
    *ALSO_INLINED = sub () { $x };
}
```

Perl 5.22 also introduces the experimental "const" attribute as an alternative. (Disable the "experimental::const_attr" warnings if you want to use it.) When applied to an anonymous subroutine, it forces the sub to be called when the `sub` expression is evaluated. The return value is captured and turned into a constant subroutine:

```
my $x = 54321;
*INLINED = sub : const { $x };
$x++;
```

The return value of `INLINED` in this example will always be 54321, regardless of later modifications to `$x`. You can also put any arbitrary code inside the sub, at it will be executed immediately and its return value captured the same way.

If you really want a subroutine with a `()` prototype that returns a lexical variable you can easily force it to not be inlined by adding an explicit `return`:

```
BEGIN {
    my $x = 79907;
    *RT_79908 = sub () { return $x };
    $x++;
}
print RT_79908(); # prints 79908
```

The easiest way to tell if a subroutine was inlined is by using B::Deparse. Consider this example of two subroutines returning `1`, one with a `()` prototype causing it to be inlined, and one without (with deparse output truncated for clarity):

```
$ perl -MO=Deparse -le 'sub ONE { 1 } if (ONE) { print ONE if ONE }'
sub ONE {
    1;
}
if (ONE ) {
    print ONE() if ONE ;
}
$ perl -MO=Deparse -le 'sub ONE () { 1 } if (ONE) { print ONE if ONE }'
sub ONE () { 1 }
do {
    print 1
};
```

If you redefine a subroutine that was eligible for inlining, you'll get a warning by default. You can use this warning to tell whether or not a particular subroutine is considered inlinable, since it's different than the warning for overriding non-inlined subroutines:

```
$ perl -e 'sub one () {1} sub one () {2}'
Constant subroutine one redefined at -e line 1.
$ perl -we 'sub one {1} sub one {2}'
Subroutine one redefined at -e line 1.
```

The warning is considered severe enough not to be affected by the **-w** switch (or its absence) because previously compiled invocations of the function will still be using the old value of the function. If you need to be able to redefine the subroutine, you need to ensure that it isn't inlined, either by dropping the `()` prototype (which changes calling semantics, so beware) or by thwarting the inlining mechanism in some

other way, e.g. by adding an explicit `return`, as mentioned above:

```
sub not_inlined () { return 23 }
```

Overriding Built-in Functions

Many built-in functions may be overridden, though this should be tried only occasionally and for good reason. Typically this might be done by a package attempting to emulate missing built-in functionality on a non-Unix system.

Overriding may be done only by importing the name from a module at compile time — ordinary predeclaration isn't good enough. However, the `use subs` pragma lets you, in effect, predeclare subs via the import syntax, and these names may then override built-in ones:

```
use subs 'chdir', 'chroot', 'chmod', 'chown';
chdir $somewhere;
sub chdir { ... }
```

To unambiguously refer to the built-in form, precede the built-in name with the special package qualifier `CORE::`. For example, saying `CORE::open()` always refers to the built-in `open()`, even if the current package has imported some other subroutine called `&open()` from elsewhere. Even though it looks like a regular function call, it isn't: the CORE:: prefix in that case is part of Perl's syntax, and works for any keyword, regardless of what is in the CORE package. Taking a reference to it, that is, `\&CORE::open`, only works for some keywords. See CORE.

Library modules should not in general export built-in names like `open` or `chdir` as part of their default `@EXPORT` list, because these may sneak into someone else's namespace and change the semantics unexpectedly. Instead, if the module adds that name to `@EXPORT_OK`, then it's possible for a user to import the name explicitly, but not implicitly. That is, they could say

```
use Module 'open';
```

and it would import the `open` override. But if they said

```
use Module;
```

they would get the default imports without overrides.

The foregoing mechanism for overriding built-in is restricted, quite deliberately, to the package that requests the import. There is a second method that is sometimes applicable when you wish to override a built-in everywhere, without regard to namespace boundaries. This is achieved by importing a sub into the special namespace `CORE::GLOBAL::`. Here is an example that quite brazenly replaces the `glob` operator with something that understands regular expressions.

```
package REGlob;
require Exporter;
@ISA = 'Exporter';
@EXPORT_OK = 'glob';

sub import {
    my $pkg = shift;
    return unless @_;
    my $sym = shift;
    my $where = ($sym =~ s/^GLOBAL_// ? 'CORE::GLOBAL' : caller(0));
    $pkg->export($where, $sym, @_);
}

sub glob {
    my $pat = shift;
    my @got;
    if (opendir my $d, '.') {
        @got = grep /$pat/, readdir $d;
        closedir $d;
    }
    return @got;
}
1;
```

And here's how it could be (ab)used:

```
#use REGlob 'GLOBAL_glob';      # override glob() in ALL namespaces
package Foo;
use REGlob 'glob';              # override glob() in Foo:: only
print for <^[a-z_]+\.pm\$>;     # show all pragmatic modules
```

The initial comment shows a contrived, even dangerous example. By overriding `glob` globally, you would be forcing the new (and subversive) behavior for the `glob` operator for *every* namespace, without the complete cognizance or cooperation of the modules that own those namespaces. Naturally, this should be done with extreme caution — if it must be done at all.

The `REGlob` example above does not implement all the support needed to cleanly override perl's `glob` operator. The built-in `glob` has different behaviors depending on whether it appears in a scalar or list context, but our `REGlob` doesn't. Indeed, many perl built-in have such context sensitive behaviors, and these must be adequately supported by a properly written override. For a fully functional example of overriding `glob`, study the implementation of `File::DosGlob` in the standard library.

When you override a built-in, your replacement should be consistent (if possible) with the built-in native syntax. You can achieve this by using a suitable prototype. To get the prototype of an overridable built-in, use the `prototype` function with an argument of `"CORE::builtin_name"` (see ''prototype'' in perlfunc).

Note however that some built-ins can't have their syntax expressed by a prototype (such as `system` or `chomp`). If you override them you won't be able to fully mimic their original syntax.

The built-ins `do`, `require` and `glob` can also be overridden, but due to special magic, their original syntax is preserved, and you don't have to define a prototype for their replacements. (You can't override the `do BLOCK` syntax, though).

`require` has special additional dark magic: if you invoke your `require` replacement as `require Foo::Bar`, it will actually receive the argument `"Foo/Bar.pm"` in `@_`. See ''require'' in perlfunc.

And, as you'll have noticed from the previous example, if you override `glob`, the `<*>` glob operator is overridden as well.

In a similar fashion, overriding the `readline` function also overrides the equivalent I/O operator `<FILEHANDLE>`. Also, overriding `readpipe` also overrides the operators `` ` ` `` and `qx//`.

Finally, some built-ins (e.g. `exists` or `grep`) can't be overridden.

Autoloading

If you call a subroutine that is undefined, you would ordinarily get an immediate, fatal error complaining that the subroutine doesn't exist. (Likewise for subroutines being used as methods, when the method doesn't exist in any base class of the class's package.) However, if an `AUTOLOAD` subroutine is defined in the package or packages used to locate the original subroutine, then that `AUTOLOAD` subroutine is called with the arguments that would have been passed to the original subroutine. The fully qualified name of the original subroutine magically appears in the global `$AUTOLOAD` variable of the same package as the `AUTOLOAD` routine. The name is not passed as an ordinary argument because, er, well, just because, that's why. (As an exception, a method call to a nonexistent `import` or `unimport` method is just skipped instead. Also, if the AUTOLOAD subroutine is an XSUB, there are other ways to retrieve the subroutine name. See ''Autoloading with XSUBs'' in perlguts for details.)

Many `AUTOLOAD` routines load in a definition for the requested subroutine using *eval()*, then execute that subroutine using a special form of *goto()* that erases the stack frame of the `AUTOLOAD` routine without a trace. (See the source to the standard module documented in AutoLoader, for example.) But an `AUTOLOAD` routine can also just emulate the routine and never define it. For example, let's pretend that a function that wasn't defined should just invoke `system` with those arguments. All you'd do is:

```
sub AUTOLOAD {
    my $program = $AUTOLOAD;
    $program =~ s/.*::/;
    system($program, @_);
}
date();
who('am', 'i');
ls('-l');
```

In fact, if you predeclare functions you want to call that way, you don't even need parentheses:

```
use subs qw(date who ls);
date;
who "am", "i";
ls '-l';
```

A more complete example of this is the Shell module on CPAN, which can treat undefined subroutine calls as calls to external programs.

Mechanisms are available to help modules writers split their modules into autoloadable files. See the standard AutoLoader module described in AutoLoader and in AutoSplit, the standard SelfLoader modules in SelfLoader, and the document on adding C functions to Perl code in perlxs.

Subroutine Attributes

A subroutine declaration or definition may have a list of attributes associated with it. If such an attribute list is present, it is broken up at space or colon boundaries and treated as though a `use attributes` had been seen. See attributes for details about what attributes are currently supported. Unlike the limitation with the obsolescent `use attrs`, the `sub : ATTRLIST` syntax works to associate the attributes with a pre-declaration, and not just with a subroutine definition.

The attributes must be valid as simple identifier names (without any punctuation other than the '_' character). They may have a parameter list appended, which is only checked for whether its parentheses ('(',')') nest properly.

Examples of valid syntax (even though the attributes are unknown):

```
sub fnord (&\%) : switch(10,foo(7,3))  :  expensive;
sub plugh () : Ugly('\(") :Bad;
sub xyzzy : _5x5 { ... }
```

Examples of invalid syntax:

```
sub fnord : switch(10,foo(); # ()-string not balanced
sub snoid : Ugly('(');        # ()-string not balanced
sub xyzzy : 5x5;              # "5x5" not a valid identifier
sub plugh : Y2::north;        # "Y2::north" not a simple identifier
sub snurt : foo + bar;        # "+" not a colon or space
```

The attribute list is passed as a list of constant strings to the code which associates them with the subroutine. In particular, the second example of valid syntax above currently looks like this in terms of how it's parsed and invoked:

```
use attributes __PACKAGE__, \&plugh, q[Ugly('\(")], 'Bad';
```

For further details on attribute lists and their manipulation, see attributes and Attribute::Handlers.

SEE ALSO

See "Function Templates" in perlref for more about references and closures. See perlxs if you'd like to learn about calling C subroutines from Perl. See perlembed if you'd like to learn about calling Perl subroutines from C. See perlmod to learn about bundling up your functions in separate files. See perlmodlib to learn what library modules come standard on your system. See perlootut to learn how to make object method calls.

NAME

perlfunc – Perl builtin functions

DESCRIPTION

The functions in this section can serve as terms in an expression. They fall into two major categories: list operators and named unary operators. These differ in their precedence relationship with a following comma. (See the precedence table in perlop.) List operators take more than one argument, while unary operators can never take more than one argument. Thus, a comma terminates the argument of a unary operator, but merely separates the arguments of a list operator. A unary operator generally provides scalar context to its argument, while a list operator may provide either scalar or list contexts for its arguments. If it does both, scalar arguments come first and list argument follow, and there can only ever be one such list argument. For instance, `splice` has three scalar arguments followed by a list, whereas `gethostbyname` has four scalar arguments.

In the syntax descriptions that follow, list operators that expect a list (and provide list context for elements of the list) are shown with LIST as an argument. Such a list may consist of any combination of scalar arguments or list values; the list values will be included in the list as if each individual element were interpolated at that point in the list, forming a longer single-dimensional list value. Commas should separate literal elements of the LIST.

Any function in the list below may be used either with or without parentheses around its arguments. (The syntax descriptions omit the parentheses.) If you use parentheses, the simple but occasionally surprising rule is this: It *looks* like a function, therefore it *is* a function, and precedence doesn't matter. Otherwise it's a list operator or unary operator, and precedence does matter. Whitespace between the function and left parenthesis doesn't count, so sometimes you need to be careful:

```
print 1+2+4;      # Prints 7.
print(1+2) + 4;   # Prints 3.
print (1+2)+4;    # Also prints 3!
print +(1+2)+4;   # Prints 7.
print ((1+2)+4);  # Prints 7.
```

If you run Perl with the `use warnings` pragma, it can warn you about this. For example, the third line above produces:

```
print (...) interpreted as function at - line 1.
Useless use of integer addition in void context at - line 1.
```

A few functions take no arguments at all, and therefore work as neither unary nor list operators. These include such functions as `time` and `endpwent`. For example, `time+86_400` always means `time() + 86_400`.

For functions that can be used in either a scalar or list context, nonabortive failure is generally indicated in scalar context by returning the undefined value, and in list context by returning the empty list.

Remember the following important rule: There is **no rule** that relates the behavior of an expression in list context to its behavior in scalar context, or vice versa. It might do two totally different things. Each operator and function decides which sort of value would be most appropriate to return in scalar context. Some operators return the length of the list that would have been returned in list context. Some operators return the first value in the list. Some operators return the last value in the list. Some operators return a count of successful operations. In general, they do what you want, unless you want consistency.

A named array in scalar context is quite different from what would at first glance appear to be a list in scalar context. You can't get a list like `(1,2,3)` into being in scalar context, because the compiler knows the context at compile time. It would generate the scalar comma operator there, not the list concatenation version of the comma. That means it was never a list to start with.

In general, functions in Perl that serve as wrappers for system calls (“syscalls”) of the same name (like *chown*(2), *fork*(2), *closedir*(2), etc.) return true when they succeed and `undef` otherwise, as is usually mentioned in the descriptions below. This is different from the C interfaces, which return `-1` on failure. Exceptions to this rule include `wait`, `waitpid`, and `syscall`. System calls also set the special `$!` variable on failure. Other functions do not, except accidentally.

Extension modules can also hook into the Perl parser to define new kinds of keyword-headed expression. These may look like functions, but may also look completely different. The syntax following the keyword is defined entirely by the extension. If you are an implementor, see “PL_keyword_plugin” in perlapi for

the mechanism. If you are using such a module, see the module's documentation for details of the syntax that it defines.

Perl Functions by Category

Here are Perl's functions (including things that look like functions, like some keywords and named operators) arranged by category. Some functions appear in more than one place.

Functions for SCALARs or strings

`chomp`, `chop`, `chr`, `crypt`, `fc`, `hex`, `index`, `lc`, `lcfirst`, `length`, `oct`, `ord`, `pack`, `q//`, `qq//`, `reverse`, `rindex`, `sprintf`, `substr`, `tr///`, `uc`, `ucfirst`, `y///`

`fc` is available only if the `"fc"` feature is enabled or if it is prefixed with `CORE::`. The `"fc"` feature is enabled automatically with a `use v5.16` (or higher) declaration in the current scope.

Regular expressions and pattern matching

`m//`, `pos`, `qr//`, `quotemeta`, `s///`, `split`, `study`

Numeric functions

`abs`, `atan2`, `cos`, `exp`, `hex`, `int`, `log`, `oct`, `rand`, `sin`, `sqrt`, `srand`

Functions for real `@ARRAY`s

`each`, `keys`, `pop`, `push`, `shift`, `splice`, `unshift`, `values`

Functions for list data

`grep`, `join`, `map`, `qw//`, `reverse`, `sort`, `unpack`

Functions for real `%HASH`es

`delete`, `each`, `exists`, `keys`, `values`

Input and output functions

`binmode`, `close`, `closedir`, `dbmclose`, `dbmopen`, `die`, `eof`, `fileno`, `flock`, `format`, `getc`, `print`, `printf`, `read`, `readdir`, `readline`, `rewinddir`, `say`, `seek`, `seekdir`, `select`, `syscall`, `sysread`, `sysseek`, `syswrite`, `tell`, `telldir`, `truncate`, `warn`, `write`

`say` is available only if the `"say"` feature is enabled or if it is prefixed with `CORE::`. The `"say"` feature is enabled automatically with a `use v5.10` (or higher) declaration in the current scope.

Functions for fixed-length data or records

`pack`, `read`, `syscall`, `sysread`, `sysseek`, `syswrite`, `unpack`, `vec`

Functions for filehandles, files, or directories

-X, `chdir`, `chmod`, `chown`, `chroot`, `fcntl`, `glob`, `ioctl`, `link`, `lstat`, `mkdir`, `open`, `opendir`, `readlink`, `rename`, `rmdir`, `select`, `stat`, `symlink`, `sysopen`, `umask`, `unlink`, `utime`

Keywords related to the control flow of your Perl program

`break`, `caller`, `continue`, `die`, `do`, `dump`, `eval`, `evalbytes`, `exit`, `__FILE__`, `goto`, `last`, `__LINE__`, `next`, `__PACKAGE__`, `redo`, `return`, `sub`, `__SUB__`, `wantarray`

`break` is available only if you enable the experimental `"switch"` feature or use the `CORE::` prefix. The `"switch"` feature also enables the `default`, `given` and `when` statements, which are documented in "Switch Statements" in perlsyn. The `"switch"` feature is enabled automatically with a `use v5.10` (or higher) declaration in the current scope. In Perl v5.14 and earlier, `continue` required the `"switch"` feature, like the other keywords.

`evalbytes` is only available with the `"evalbytes"` feature (see feature) or if prefixed with `CORE::`. `__SUB__` is only available with the `"current_sub"` feature or if prefixed with `CORE::`. Both the `"evalbytes"` and `"current_sub"` features are enabled automatically with a `use v5.16` (or higher) declaration in the current scope.

Keywords related to scoping

`caller`, `import`, `local`, `my`, `our`, `package`, `state`, `use`

`state` is available only if the `"state"` feature is enabled or if it is prefixed with `CORE::`. The `"state"` feature is enabled automatically with a `use v5.10` (or higher) declaration in the current scope.

Miscellaneous functions
: `defined`, `formline`, `lock`, `prototype`, `reset`, `scalar`, `undef`

Functions for processes and process groups
: `alarm`, `exec`, `fork`, `getpgrp`, `getppid`, `getpriority`, `kill`, `pipe`, `qx//`, `readpipe`, `setpgrp`, `setpriority`, `sleep`, `system`, `times`, `wait`, `waitpid`

Keywords related to Perl modules
: `do`, `import`, `no`, `package`, `require`, `use`

Keywords related to classes and object-orientation
: `bless`, `dbmclose`, `dbmopen`, `package`, `ref`, `tie`, `tied`, `untie`, `use`

Low-level socket functions
: `accept`, `bind`, `connect`, `getpeername`, `getsockname`, `getsockopt`, `listen`, `recv`, `send`, `setsockopt`, `shutdown`, `socket`, `socketpair`

System V interprocess communication functions
: `msgctl`, `msgget`, `msgrcv`, `msgsnd`, `semctl`, `semget`, `semop`, `shmctl`, `shmget`, `shmread`, `shmwrite`

Fetching user and group info
: `endgrent`, `endhostent`, `endnetent`, `endpwent`, `getgrent`, `getgrgid`, `getgrnam`, `getlogin`, `getpwent`, `getpwnam`, `getpwuid`, `setgrent`, `setpwent`

Fetching network info
: `endprotoent`, `endservent`, `gethostbyaddr`, `gethostbyname`, `gethostent`, `getnetbyaddr`, `getnetbyname`, `getnetent`, `getprotobyname`, `getprotobynumber`, `getprotoent`, `getservbyname`, `getservbyport`, `getservent`, `sethostent`, `setnetent`, `setprotoent`, `setservent`

Time-related functions
: `gmtime`, `localtime`, `time`, `times`

Non-function keywords
: `and`, `AUTOLOAD`, `BEGIN`, `CHECK`, `cmp`, `CORE`, `__DATA__`, `default`, `DESTROY`, `else`, `elseif`, `elsif`, `END`, `__END__`, `eq`, `for`, `foreach`, `ge`, `given`, `gt`, `if`, `INIT`, `le`, `lt`, `ne`, `not`, `or`, `UNITCHECK`, `unless`, `until`, `when`, `while`, `x`, `xor`

Portability

Perl was born in Unix and can therefore access all common Unix system calls. In non-Unix environments, the functionality of some Unix system calls may not be available or details of the available functionality may differ slightly. The Perl functions affected by this are:

`-X`, `binmode`, `chmod`, `chown`, `chroot`, `crypt`, `dbmclose`, `dbmopen`, `dump`, `endgrent`, `endhostent`, `endnetent`, `endprotoent`, `endpwent`, `endservent`, `exec`, `fcntl`, `flock`, `fork`, `getgrent`, `getgrgid`, `gethostbyname`, `gethostent`, `getlogin`, `getnetbyaddr`, `getnetbyname`, `getnetent`, `getppid`, `getpgrp`, `getpriority`, `getprotobynumber`, `getprotoent`, `getpwent`, `getpwnam`, `getpwuid`, `getservbyport`, `getservent`, `getsockopt`, `glob`, `ioctl`, `kill`, `link`, `lstat`, `msgctl`, `msgget`, `msgrcv`, `msgsnd`, `open`, `pipe`, `readlink`, `rename`, `select`, `semctl`, `semget`, `semop`, `setgrent`, `sethostent`, `setnetent`, `setpgrp`, `setpriority`, `setprotoent`, `setpwent`, `setservent`, `setsockopt`, `shmctl`, `shmget`, `shmread`, `shmwrite`, `socket`, `socketpair`, `stat`, `symlink`, `syscall`, `sysopen`, `system`, `times`, `truncate`, `umask`, `unlink`, `utime`, `wait`, `waitpid`

For more information about the portability of these functions, see perlport and other available platform-specific documentation.

Alphabetical Listing of Perl Functions

-X FILEHANDLE

-X EXPR

-X DIRHANDLE

-X A file test, where X is one of the letters listed below. This unary operator takes one argument, either a filename, a filehandle, or a dirhandle, and tests the associated file to see if something is true about it. If the argument is omitted, tests `$_`, except for `-t`, which tests STDIN. Unless otherwise documented, it returns `1` for true and `''` for false. If the file doesn't exist or can't be examined, it returns `undef`

and sets `$!` (errno). Despite the funny names, precedence is the same as any other named unary operator. The operator may be any of:

```
-r  File is readable by effective uid/gid.
-w  File is writable by effective uid/gid.
-x  File is executable by effective uid/gid.
-o  File is owned by effective uid.

-R  File is readable by real uid/gid.
-W  File is writable by real uid/gid.
-X  File is executable by real uid/gid.
-O  File is owned by real uid.

-e  File exists.
-z  File has zero size (is empty).
-s  File has nonzero size (returns size in bytes).

-f  File is a plain file.
-d  File is a directory.
-l  File is a symbolic link (false if symlinks aren't
    supported by the file system).
-p  File is a named pipe (FIFO), or Filehandle is a pipe.
-S  File is a socket.
-b  File is a block special file.
-c  File is a character special file.
-t  Filehandle is opened to a tty.

-u  File has setuid bit set.
-g  File has setgid bit set.
-k  File has sticky bit set.

-T  File is an ASCII or UTF-8 text file (heuristic guess).
-B  File is a "binary" file (opposite of -T).

-M  Script start time minus file modification time, in days.
-A  Same for access time.
-C  Same for inode change time (Unix, may differ for other
    platforms)
```

Example:

```
while (<>) {
    chomp;
    next unless -f $_;  # ignore specials
    #...
}
```

Note that `-s/a/b/` does not do a negated substitution. Saying `-exp($foo)` still works as expected, however: only single letters following a minus are interpreted as file tests.

These operators are exempt from the ‘‘looks like a function rule’’ described above. That is, an opening parenthesis after the operator does not affect how much of the following code constitutes the argument. Put the opening parentheses before the operator to separate it from code that follows (this applies only to operators with higher precedence than unary operators, of course):

```
-s($file) + 1024   # probably wrong; same as -s($file + 1024)
(-s $file) + 1024  # correct
```

The interpretation of the file permission operators `-r`, `-R`, `-w`, `-W`, `-x`, and `-X` is by default based solely on the mode of the file and the uids and gids of the user. There may be other reasons you can’t actually read, write, or execute the file: for example network filesystem access controls, ACLs (access control lists), read-only filesystems, and unrecognized executable formats. Note that the use of these

six specific operators to verify if some operation is possible is usually a mistake, because it may be open to race conditions.

Also note that, for the superuser on the local filesystems, the -r, -R, -w, and -W tests always return 1, and -x and -X return 1 if any execute bit is set in the mode. Scripts run by the superuser may thus need to do a stat to determine the actual mode of the file, or temporarily set their effective uid to something else.

If you are using ACLs, there is a pragma called filetest that may produce more accurate results than the bare stat mode bits. When under use filetest 'access', the above-mentioned filetests test whether the permission can(not) be granted using the *access*(2) family of system calls. Also note that the -x and -X tests may under this pragma return true even if there are no execute permission bits set (nor any extra execute permission ACLs). This strangeness is due to the underlying system calls' definitions. Note also that, due to the implementation of use filetest 'access', the _ special filehandle won't cache the results of the file tests when this pragma is in effect. Read the documentation for the filetest pragma for more information.

The -T and -B tests work as follows. The first block or so of the file is examined to see if it is valid UTF-8 that includes non-ASCII characters. If so, it's a -T file. Otherwise, that same portion of the file is examined for odd characters such as strange control codes or characters with the high bit set. If more than a third of the characters are strange, it's a -B file; otherwise it's a -T file. Also, any file containing a zero byte in the examined portion is considered a binary file. (If executed within the scope of a use locale which includes LC_CTYPE, odd characters are anything that isn't a printable nor space in the current locale.) If -T or -B is used on a filehandle, the current IO buffer is examined rather than the first block. Both -T and -B return true on an empty file, or a file at EOF when testing a filehandle. Because you have to read a file to do the -T test, on most occasions you want to use a -f against the file first, as in next unless -f $file && -T $file.

If any of the file tests (or either the stat or lstat operator) is given the special filehandle consisting of a solitary underline, then the stat structure of the previous file test (or stat operator) is used, saving a system call. (This doesn't work with -t, and you need to remember that lstat and -l leave values in the stat structure for the symbolic link, not the real file.) (Also, if the stat buffer was filled by an lstat call, -T and -B will reset it with the results of stat _). Example:

```
print "Can do.\n" if -r $a || -w _ || -x _;

stat($filename);
print "Readable\n" if -r _;
print "Writable\n" if -w _;
print "Executable\n" if -x _;
print "Setuid\n" if -u _;
print "Setgid\n" if -g _;
print "Sticky\n" if -k _;
print "Text\n" if -T _;
print "Binary\n" if -B _;
```

As of Perl 5.10.0, as a form of purely syntactic sugar, you can stack file test operators, in a way that -f -w -x $file is equivalent to -x $file && -w _ && -f _. (This is only fancy syntax: if you use the return value of -f $file as an argument to another filetest operator, no special magic will happen.)

Portability issues: "-X" in perlport.

To avoid confusing would-be users of your code with mysterious syntax errors, put something like this at the top of your script:

```
use 5.010;  # so filetest ops can stack
```

abs VALUE

abs Returns the absolute value of its argument. If VALUE is omitted, uses $_.

accept NEWSOCKET,GENERICSOCKET

Accepts an incoming socket connect, just as *accept*(2) does. Returns the packed address if it succeeded, false otherwise. See the example in "Sockets: Client/Server Communication" in perlipc.

On systems that support a close-on-exec flag on files, the flag will be set for the newly opened file descriptor, as determined by the value of `$^F`. See "$^F" in perlvar.

alarm SECONDS
alarm

Arranges to have a SIGALRM delivered to this process after the specified number of wallclock seconds has elapsed. If SECONDS is not specified, the value stored in `$_` is used. (On some machines, unfortunately, the elapsed time may be up to one second less or more than you specified because of how seconds are counted, and process scheduling may delay the delivery of the signal even further.)

Only one timer may be counting at once. Each call disables the previous timer, and an argument of `0` may be supplied to cancel the previous timer without starting a new one. The returned value is the amount of time remaining on the previous timer.

For delays of finer granularity than one second, the Time::HiRes module (from CPAN, and starting from Perl 5.8 part of the standard distribution) provides `ualarm`. You may also use Perl's four-argument version of `select` leaving the first three arguments undefined, or you might be able to use the `syscall` interface to access *setitimer*(2) if your system supports it. See perlfaq8 for details.

It is usually a mistake to intermix `alarm` and `sleep` calls, because `sleep` may be internally implemented on your system with `alarm`.

If you want to use `alarm` to time out a system call you need to use an `eval`/`die` pair. You can't rely on the alarm causing the system call to fail with `$!` set to `EINTR` because Perl sets up signal handlers to restart system calls on some systems. Using `eval`/`die` always works, modulo the caveats given in "Signals" in perlipc.

```
eval {
    local $SIG{ALRM} = sub { die "alarm\n" }; # NB: \n required
    alarm $timeout;
    my $nread = sysread $socket, $buffer, $size;
    alarm 0;
};
if ($@) {
    die unless $@ eq "alarm\n";   # propagate unexpected errors
    # timed out
}
else {
    # didn't
}
```

For more information see perlipc.

Portability issues: "alarm" in perlport.

atan2 Y,X

Returns the arctangent of Y/X in the range −PI to PI.

For the tangent operation, you may use the `Math::Trig::tan` function, or use the familiar relation:

```
sub tan { sin($_[0]) / cos($_[0])  }
```

The return value for `atan2(0,0)` is implementation-defined; consult your *atan2*(3) manpage for more information.

Portability issues: "atan2" in perlport.

bind SOCKET,NAME

Binds a network address to a socket, just as *bind*(2) does. Returns true if it succeeded, false otherwise. NAME should be a packed address of the appropriate type for the socket. See the examples in "Sockets: Client/Server Communication" in perlipc.

binmode FILEHANDLE, LAYER
binmode FILEHANDLE

Arranges for FILEHANDLE to be read or written in "binary" or "text" mode on systems where the run-time libraries distinguish between binary and text files. If FILEHANDLE is an expression, the

value is taken as the name of the filehandle. Returns true on success, otherwise it returns `undef` and sets `$!` (errno).

On some systems (in general, DOS- and Windows-based systems) `binmode` is necessary when you're not working with a text file. For the sake of portability it is a good idea always to use it when appropriate, and never to use it when it isn't appropriate. Also, people can set their I/O to be by default UTF8-encoded Unicode, not bytes.

In other words: regardless of platform, use `binmode` on binary data, like images, for example.

If LAYER is present it is a single string, but may contain multiple directives. The directives alter the behaviour of the filehandle. When LAYER is present, using binmode on a text file makes sense.

If LAYER is omitted or specified as `:raw` the filehandle is made suitable for passing binary data. This includes turning off possible CRLF translation and marking it as bytes (as opposed to Unicode characters). Note that, despite what may be implied in *"Programming Perl"* (the Camel, 3rd edition) or elsewhere, `:raw` is *not* simply the inverse of `:crlf`. Other layers that would affect the binary nature of the stream are *also* disabled. See PerlIO, perlrun, and the discussion about the PERLIO environment variable.

The `:bytes`, `:crlf`, `:utf8`, and any other directives of the form `:...`, are called I/O *layers*. The open pragma can be used to establish default I/O layers.

The LAYER parameter of the `binmode` function is described as "DISCIPLINE" in "Programming Perl, 3rd Edition". However, since the publishing of this book, by many known as "Camel III", the consensus of the naming of this functionality has moved from "discipline" to "layer". All documentation of this version of Perl therefore refers to "layers" rather than to "disciplines". Now back to the regularly scheduled documentation...

To mark FILEHANDLE as UTF-8, use `:utf8` or `:encoding(UTF-8)`. `:utf8` just marks the data as UTF-8 without further checking, while `:encoding(UTF-8)` checks the data for actually being valid UTF-8. More details can be found in PerlIO::encoding.

In general, `binmode` should be called after `open` but before any I/O is done on the filehandle. Calling `binmode` normally flushes any pending buffered output data (and perhaps pending input data) on the handle. An exception to this is the `:encoding` layer that changes the default character encoding of the handle. The `:encoding` layer sometimes needs to be called in mid-stream, and it doesn't flush the stream. `:encoding` also implicitly pushes on top of itself the `:utf8` layer because internally Perl operates on UTF8-encoded Unicode characters.

The operating system, device drivers, C libraries, and Perl run-time system all conspire to let the programmer treat a single character (`\n`) as the line terminator, irrespective of external representation. On many operating systems, the native text file representation matches the internal representation, but on some platforms the external representation of `\n` is made up of more than one character.

All variants of Unix, Mac OS (old and new), and Stream_LF files on VMS use a single character to end each line in the external representation of text (even though that single character is CARRIAGE RETURN on old, pre-Darwin flavors of Mac OS, and is LINE FEED on Unix and most VMS files). In other systems like OS/2, DOS, and the various flavors of MS-Windows, your program sees a `\n` as a simple `\cJ`, but what's stored in text files are the two characters `\cM\cJ`. That means that if you don't use `binmode` on these systems, `\cM\cJ` sequences on disk will be converted to `\n` on input, and any `\n` in your program will be converted back to `\cM\cJ` on output. This is what you want for text files, but it can be disastrous for binary files.

Another consequence of using `binmode` (on some systems) is that special end-of-file markers will be seen as part of the data stream. For systems from the Microsoft family this means that, if your binary data contain `\cZ`, the I/O subsystem will regard it as the end of the file, unless you use `binmode`.

`binmode` is important not only for `readline` and `print` operations, but also when using `read`, `seek`, `sysread`, `syswrite` and `tell` (see perlport for more details). See the `$/` and `$\` variables in perlvar for how to manually set your input and output line-termination sequences.

Portability issues: "binmode" in perlport.

bless REF,CLASSNAME
bless REF
: This function tells the thingy referenced by REF that it is now an object in the CLASSNAME package. If CLASSNAME is omitted, the current package is used. Because a `bless` is often the last thing in a constructor, it returns the reference for convenience. Always use the two-argument version if a derived class might inherit the method doing the blessing. See perlobj for more about the blessing (and blessings) of objects.

  Consider always blessing objects in CLASSNAMEs that are mixed case. Namespaces with all lowercase names are considered reserved for Perl pragmas. Builtin types have all uppercase names. To prevent confusion, you may wish to avoid such package names as well. Make sure that CLASSNAME is a true value.

  See "Perl Modules" in perlmod.

break
: Break out of a `given` block.

  `break` is available only if the `"switch"` feature is enabled or if it is prefixed with `CORE::`. The `"switch"` feature is enabled automatically with a `use v5.10` (or higher) declaration in the current scope.

caller EXPR
caller
: Returns the context of the current pure perl subroutine call. In scalar context, returns the caller's package name if there *is* a caller (that is, if we're in a subroutine or `eval` or `require`) and the undefined value otherwise. caller never returns XS subs and they are skipped. The next pure perl sub will appear instead of the XS sub in caller's return values. In list context, caller returns

```
    # 0         1          2
  my ($package, $filename, $line) = caller;
```

  With EXPR, it returns some extra information that the debugger uses to print a stack trace. The value of EXPR indicates how many call frames to go back before the current one.

```
    #  0         1          2      3            4
 my ($package, $filename, $line, $subroutine, $hasargs,

    #  5          6          7            8       9         10
    $wantarray, $evaltext, $is_require, $hints, $bitmask, $hinthash)
  = caller($i);
```

  Here, `$subroutine` is the function that the caller called (rather than the function containing the caller). Note that `$subroutine` may be `(eval)` if the frame is not a subroutine call, but an `eval`. In such a case additional elements `$evaltext` and `$is_require` are set: `$is_require` is true if the frame is created by a `require` or `use` statement, `$evaltext` contains the text of the `eval EXPR` statement. In particular, for an `eval BLOCK` statement, `$subroutine` is `(eval)`, but `$evaltext` is undefined. (Note also that each `use` statement creates a `require` frame inside an `eval EXPR` frame.) `$subroutine` may also be `(unknown)` if this particular subroutine happens to have been deleted from the symbol table. `$hasargs` is true if a new instance of `@_` was set up for the frame. `$hints` and `$bitmask` contain pragmatic hints that the caller was compiled with. `$hints` corresponds to `$^H`, and `$bitmask` corresponds to `${^WARNING_BITS}`. The `$hints` and `$bitmask` values are subject to change between versions of Perl, and are not meant for external use.

  `$hinthash` is a reference to a hash containing the value of `%^H` when the caller was compiled, or `undef` if `%^H` was empty. Do not modify the values of this hash, as they are the actual values stored in the optree.

  Furthermore, when called from within the DB package in list context, and with an argument, caller returns more detailed information: it sets the list variable `@DB::args` to be the arguments with which the subroutine was invoked.

  Be aware that the optimizer might have optimized call frames away before `caller` had a chance to get the information. That means that `caller(N)` might not return information about the call frame

you expect it to, for `N > 1`. In particular, `@DB::args` might have information from the previous time `caller` was called.

Be aware that setting `@DB::args` is *best effort*, intended for debugging or generating backtraces, and should not be relied upon. In particular, as `@_` contains aliases to the caller's arguments, Perl does not take a copy of `@_`, so `@DB::args` will contain modifications the subroutine makes to `@_` or its contents, not the original values at call time. `@DB::args`, like `@_`, does not hold explicit references to its elements, so under certain cases its elements may have become freed and reallocated for other variables or temporary values. Finally, a side effect of the current implementation is that the effects of `shift @_` can *normally* be undone (but not `pop @_` or other splicing, *and* not if a reference to `@_` has been taken, *and* subject to the caveat about reallocated elements), so `@DB::args` is actually a hybrid of the current state and initial state of `@_`. Buyer beware.

chdir EXPR
chdir FILEHANDLE
chdir DIRHANDLE
chdir

Changes the working directory to EXPR, if possible. If EXPR is omitted, changes to the directory specified by `$ENV{HOME}`, if set; if not, changes to the directory specified by `$ENV{LOGDIR}`. (Under VMS, the variable `$ENV{'SYS$LOGIN'}` is also checked, and used if it is set.) If neither is set, `chdir` does nothing and fails. It returns true on success, false otherwise. See the example under `die`.

On systems that support *fchdir*(2), you may pass a filehandle or directory handle as the argument. On systems that don't support *fchdir*(2), passing handles raises an exception.

chmod LIST

Changes the permissions of a list of files. The first element of the list must be the numeric mode, which should probably be an octal number, and which definitely should *not* be a string of octal digits: `0644` is okay, but `"0644"` is not. Returns the number of files successfully changed. See also `oct` if all you have is a string.

```
my $cnt = chmod 0755, "foo", "bar";
chmod 0755, @executables;
my $mode = "0644"; chmod $mode, "foo";      # !!! sets mode to
                                            # --w----r-T
my $mode = "0644"; chmod oct($mode), "foo"; # this is better
my $mode = 0644;   chmod $mode, "foo";      # this is best
```

On systems that support *fchmod*(2), you may pass filehandles among the files. On systems that don't support *fchmod*(2), passing filehandles raises an exception. Filehandles must be passed as globs or glob references to be recognized; barewords are considered filenames.

```
open(my $fh, "<", "foo");
my $perm = (stat $fh)[2] & 07777;
chmod($perm | 0600, $fh);
```

You can also import the symbolic `S_I*` constants from the `Fcntl` module:

```
use Fcntl qw( :mode );
chmod S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH, @executables;
# Identical to the chmod 0755 of the example above.
```

Portability issues: "chmod" in perlport.

chomp VARIABLE
chomp(LIST)
chomp

This safer version of `chop` removes any trailing string that corresponds to the current value of `$/` (also known as `$INPUT_RECORD_SEPARATOR` in the `English` module). It returns the total number of characters removed from all its arguments. It's often used to remove the newline from the end of an input record when you're worried that the final record may be missing its newline. When in paragraph mode (`$/ = ''`), it removes all trailing newlines from the string. When in slurp mode (`$/ = undef`) or fixed-length record mode (`$/` is a reference to an integer or the like; see perlvar),

`chomp` won't remove anything. If VARIABLE is omitted, it chomps `$_`. Example:

```
while (<>) {
    chomp;  # avoid \n on last field
    my @array = split(/:/);
    # ...
}
```

If VARIABLE is a hash, it chomps the hash's values, but not its keys, resetting the `each` iterator in the process.

You can actually chomp anything that's an lvalue, including an assignment:

```
chomp(my $cwd = `pwd`);
chomp(my $answer = <STDIN>);
```

If you chomp a list, each element is chomped, and the total number of characters removed is returned.

Note that parentheses are necessary when you're chomping anything that is not a simple variable. This is because `chomp $cwd = `pwd`;` is interpreted as `(chomp $cwd) = `pwd`;`, rather than as `chomp( $cwd = `pwd` )` which you might expect. Similarly, `chomp $a, $b` is interpreted as `chomp($a), $b` rather than as `chomp($a, $b)`.

chop VARIABLE
chop(LIST)
chop

Chops off the last character of a string and returns the character chopped. It is much more efficient than `s/.$//s` because it neither scans nor copies the string. If VARIABLE is omitted, chops `$_`. If VARIABLE is a hash, it chops the hash's values, but not its keys, resetting the `each` iterator in the process.

You can actually chop anything that's an lvalue, including an assignment.

If you chop a list, each element is chopped. Only the value of the last `chop` is returned.

Note that `chop` returns the last character. To return all but the last character, use `substr($string, 0, -1)`.

See also `chomp`.

chown LIST

Changes the owner (and group) of a list of files. The first two elements of the list must be the *numeric* uid and gid, in that order. A value of −1 in either position is interpreted by most systems to leave that value unchanged. Returns the number of files successfully changed.

```
my $cnt = chown $uid, $gid, 'foo', 'bar';
chown $uid, $gid, @filenames;
```

On systems that support *fchown*(2), you may pass filehandles among the files. On systems that don't support *fchown*(2), passing filehandles raises an exception. Filehandles must be passed as globs or glob references to be recognized; barewords are considered filenames.

Here's an example that looks up nonnumeric uids in the passwd file:

```
print "User: ";
chomp(my $user = <STDIN>);
print "Files: ";
chomp(my $pattern = <STDIN>);

my ($login,$pass,$uid,$gid) = getpwnam($user)
    or die "$user not in passwd file";

my @ary = glob($pattern);  # expand filenames
chown $uid, $gid, @ary;
```

On most systems, you are not allowed to change the ownership of the file unless you're the superuser, although you should be able to change the group to any of your secondary groups. On insecure

systems, these restrictions may be relaxed, but this is not a portable assumption. On POSIX systems, you can detect this condition this way:

```
use POSIX qw(sysconf _PC_CHOWN_RESTRICTED);
my $can_chown_giveaway = ! sysconf(_PC_CHOWN_RESTRICTED);
```

Portability issues: "chown" in perlport.

chr NUMBER

chr Returns the character represented by that NUMBER in the character set. For example, `chr(65)` is `"A"` in either ASCII or Unicode, and chr(0x263a) is a Unicode smiley face.

Negative values give the Unicode replacement character (*chr*(0xfffd)), except under the bytes pragma, where the low eight bits of the value (truncated to an integer) are used.

If NUMBER is omitted, uses `$_`.

For the reverse, use `ord`.

Note that characters from 128 to 255 (inclusive) are by default internally not encoded as UTF-8 for backward compatibility reasons.

See perlunicode for more about Unicode.

chroot FILENAME

chroot

This function works like the system call by the same name: it makes the named directory the new root directory for all further pathnames that begin with a `/` by your process and all its children. (It doesn't change your current working directory, which is unaffected.) For security reasons, this call is restricted to the superuser. If FILENAME is omitted, does a `chroot` to `$_`.

NOTE: It is good security practice to do `chdir("/")` (`chdir` to the root directory) immediately after a `chroot`.

Portability issues: "chroot" in perlport.

close FILEHANDLE

close

Closes the file or pipe associated with the filehandle, flushes the IO buffers, and closes the system file descriptor. Returns true if those operations succeed and if no error was reported by any PerlIO layer. Closes the currently selected filehandle if the argument is omitted.

You don't have to close FILEHANDLE if you are immediately going to do another `open` on it, because `open` closes it for you. (See `open`.) However, an explicit `close` on an input file resets the line counter (`$.`), while the implicit close done by `open` does not.

If the filehandle came from a piped open, `close` returns false if one of the other syscalls involved fails or if its program exits with non-zero status. If the only problem was that the program exited non-zero, `$!` will be set to `0`. Closing a pipe also waits for the process executing on the pipe to exit — in case you wish to look at the output of the pipe afterwards — and implicitly puts the exit status value of that command into `$?` and `${^CHILD_ERROR_NATIVE}`.

If there are multiple threads running, `close` on a filehandle from a piped open returns true without waiting for the child process to terminate, if the filehandle is still open in another thread.

Closing the read end of a pipe before the process writing to it at the other end is done writing results in the writer receiving a SIGPIPE. If the other end can't handle that, be sure to read all the data before closing the pipe.

Example:

```
open(OUTPUT, '|sort >foo')  # pipe to sort
    or die "Can't start sort: $!";
#...                        # print stuff to output
close OUTPUT                # wait for sort to finish
    or warn $! ? "Error closing sort pipe: $!"
               : "Exit status $? from sort";
open(INPUT, 'foo')          # get sort's results
    or die "Can't open 'foo' for input: $!";
```

FILEHANDLE may be an expression whose value can be used as an indirect filehandle, usually the real filehandle name or an autovivified handle.

closedir DIRHANDLE
: Closes a directory opened by `opendir` and returns the success of that system call.

connect SOCKET,NAME
: Attempts to connect to a remote socket, just like *connect*(2). Returns true if it succeeded, false otherwise. NAME should be a packed address of the appropriate type for the socket. See the examples in ''Sockets: Client/Server Communication'' in perlipc.

continue BLOCK
continue
: When followed by a BLOCK, `continue` is actually a flow control statement rather than a function. If there is a `continue` BLOCK attached to a BLOCK (typically in a `while` or `foreach`), it is always executed just before the conditional is about to be evaluated again, just like the third part of a `for` loop in C. Thus it can be used to increment a loop variable, even when the loop has been continued via the `next` statement (which is similar to the C `continue` statement).

  `last`, `next`, or `redo` may appear within a `continue` block; `last` and `redo` behave as if they had been executed within the main block. So will `next`, but since it will execute a `continue` block, it may be more entertaining.

```
while (EXPR) {
    ### redo always comes here
    do_something;
} continue {
    ### next always comes here
    do_something_else;
    # then back the top to re-check EXPR
}
### last always comes here
```

  Omitting the `continue` section is equivalent to using an empty one, logically enough, so `next` goes directly back to check the condition at the top of the loop.

  When there is no BLOCK, `continue` is a function that falls through the current `when` or `default` block instead of iterating a dynamically enclosing `foreach` or exiting a lexically enclosing `given`. In Perl 5.14 and earlier, this form of `continue` was only available when the `"switch"` feature was enabled. See feature and ''Switch Statements'' in perlsyn for more information.

cos EXPR
cos
: Returns the cosine of EXPR (expressed in radians). If EXPR is omitted, takes the cosine of `$_`.

  For the inverse cosine operation, you may use the `Math::Trig::acos` function, or use this relation:

```
sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }
```

crypt PLAINTEXT,SALT
: Creates a digest string exactly like the *crypt*(3) function in the C library (assuming that you actually have a version there that has not been extirpated as a potential munition).

  `crypt` is a one-way hash function. The PLAINTEXT and SALT are turned into a short string, called a digest, which is returned. The same PLAINTEXT and SALT will always return the same string, but there is no (known) way to get the original PLAINTEXT from the hash. Small changes in the PLAINTEXT or SALT will result in large changes in the digest.

There is no decrypt function. This function isn't all that useful for cryptography (for that, look for *Crypt* modules on your nearby CPAN mirror) and the name "crypt" is a bit of a misnomer. Instead it is primarily used to check if two pieces of text are the same without having to transmit or store the text itself. An example is checking if a correct password is given. The digest of the password is stored, not the password itself. The user types in a password that is `crypt`'d with the same salt as the stored digest. If the two digests match, the password is correct.

When verifying an existing digest string you should use the digest as the salt (like `crypt($plain, $digest) eq $digest`). The SALT used to create the digest is visible as part of the digest. This ensures `crypt` will hash the new string with the same salt as the digest. This allows your code to work with the standard `crypt` and with more exotic implementations. In other words, assume nothing about the returned string itself nor about how many bytes of SALT may matter.

Traditionally the result is a string of 13 bytes: two first bytes of the salt, followed by 11 bytes from the set `[./0-9A-Za-z]`, and only the first eight bytes of PLAINTEXT mattered. But alternative hashing schemes (like MD5), higher level security schemes (like C2), and implementations on non-Unix platforms may produce different strings.

When choosing a new salt create a random two character string whose characters come from the set `[./0-9A-Za-z]` (like `join '', ('.', '/', 0..9, 'A'..'Z', 'a'..'z')[rand 64, rand 64]`). This set of characters is just a recommendation; the characters allowed in the salt depend solely on your system's crypt library, and Perl can't restrict what salts `crypt` accepts.

Here's an example that makes sure that whoever runs this program knows their password:

```
my $pwd = (getpwuid($<))[1];

system "stty -echo";
print "Password: ";
chomp(my $word = <STDIN>);
print "\n";
system "stty echo";

if (crypt($word, $pwd) ne $pwd) {
    die "Sorry...\n";
} else {
    print "ok\n";
}
```

Of course, typing in your own password to whoever asks you for it is unwise.

The `crypt` function is unsuitable for hashing large quantities of data, not least of all because you can't get the information back. Look at the Digest module for more robust algorithms.

If using `crypt` on a Unicode string (which *potentially* has characters with codepoints above 255), Perl tries to make sense of the situation by trying to downgrade (a copy of) the string back to an eight-bit byte string before calling `crypt` (on that copy). If that works, good. If not, `crypt` dies with `Wide character in crypt`.

Portability issues: "crypt" in perlport.

dbmclose HASH

[This function has been largely superseded by the `untie` function.]

Breaks the binding between a DBM file and a hash.

Portability issues: "dbmclose" in perlport.

dbmopen HASH,DBNAME,MASK

[This function has been largely superseded by the `tie` function.]

This binds a *dbm*(3), *ndbm*(3), *sdbm*(3), *gdbm*(3), or Berkeley DB file to a hash. HASH is the name of the hash. (Unlike normal `open`, the first argument is *not* a filehandle, even though it looks like one). DBNAME is the name of the database (without the *.dir* or *.pag* extension if any). If the database does not exist, it is created with protection specified by MASK (as modified by the `umask`). To prevent creation of the database if it doesn't exist, you may specify a MODE of 0, and the function will

return a false value if it can't find an existing database. If your system supports only the older DBM functions, you may make only one `dbmopen` call in your program. In older versions of Perl, if your system had neither DBM nor ndbm, calling `dbmopen` produced a fatal error; it now falls back to *sdbm*(3).

If you don't have write access to the DBM file, you can only read hash variables, not set them. If you want to test whether you can write, either use file tests or try setting a dummy hash entry inside an `eval` to trap the error.

Note that functions such as `keys` and `values` may return huge lists when used on large DBM files. You may prefer to use the `each` function to iterate over large DBM files. Example:

```
# print out history file offsets
dbmopen(%HIST,'/usr/lib/news/history',0666);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
dbmclose(%HIST);
```

See also AnyDBM_File for a more general description of the pros and cons of the various dbm approaches, as well as DB_File for a particularly rich implementation.

You can control which DBM library you use by loading that library before you call `dbmopen`:

```
use DB_File;
dbmopen(%NS_Hist, "$ENV{HOME}/.netscape/history.db")
    or die "Can't open netscape history file: $!";
```

Portability issues: "dbmopen" in perlport.

defined EXPR
defined

Returns a Boolean value telling whether EXPR has a value other than the undefined value `undef`. If EXPR is not present, `$_` is checked.

Many operations return `undef` to indicate failure, end of file, system error, uninitialized variable, and other exceptional conditions. This function allows you to distinguish `undef` from other values. (A simple Boolean test will not distinguish among `undef`, zero, the empty string, and `"0"`, which are all equally false.) Note that since `undef` is a valid scalar, its presence doesn't *necessarily* indicate an exceptional condition: `pop` returns `undef` when its argument is an empty array, *or* when the element to return happens to be `undef`.

You may also use `defined(&func)` to check whether subroutine `func` has ever been defined. The return value is unaffected by any forward declarations of `func`. A subroutine that is not defined may still be callable: its package may have an `AUTOLOAD` method that makes it spring into existence the first time that it is called; see perlsub.

Use of `defined` on aggregates (hashes and arrays) is no longer supported. It used to report whether memory for that aggregate had ever been allocated. You should instead use a simple test for size:

```
if (@an_array) { print "has array elements\n" }
if (%a_hash)   { print "has hash members\n"   }
```

When used on a hash element, it tells you whether the value is defined, not whether the key exists in the hash. Use `exists` for the latter purpose.

Examples:

```
print if defined $switch{D};
print "$val\n" while defined($val = pop(@ary));
die "Can't readlink $sym: $!"
    unless defined($value = readlink $sym);
sub foo { defined &$bar ? $bar->(@_) : die "No bar"; }
$debugging = 0 unless defined $debugging;
```

Note: Many folks tend to overuse `defined` and are then surprised to discover that the number `0` and `""` (the zero-length string) are, in fact, defined values. For example, if you say

```
"ab" =~ /a(.*)b/;
```

The pattern match succeeds and `$1` is defined, although it matched "nothing". It didn't really fail to match anything. Rather, it matched something that happened to be zero characters long. This is all very above-board and honest. When a function returns an undefined value, it's an admission that it couldn't give you an honest answer. So you should use `defined` only when questioning the integrity of what you're trying to do. At other times, a simple comparison to `0` or `""` is what you want.

See also `undef`, `exists`, `ref`.

delete EXPR

Given an expression that specifies an element or slice of a hash, `delete` deletes the specified elements from that hash so that `exists` on that element no longer returns true. Setting a hash element to the undefined value does not remove its key, but deleting it does; see `exists`.

In list context, returns the value or values deleted, or the last such element in scalar context. The return list's length always matches that of the argument list: deleting non-existent elements returns the undefined value in their corresponding positions.

`delete` may also be used on arrays and array slices, but its behavior is less straightforward. Although `exists` will return false for deleted entries, deleting array elements never changes indices of existing values; use `shift` or `splice` for that. However, if any deleted elements fall at the end of an array, the array's size shrinks to the position of the highest element that still tests true for `exists`, or to 0 if none do. In other words, an array won't have trailing nonexistent elements after a delete.

WARNING: Calling `delete` on array values is strongly discouraged. The notion of deleting or checking the existence of Perl array elements is not conceptually coherent, and can lead to surprising behavior.

Deleting from `%ENV` modifies the environment. Deleting from a hash tied to a DBM file deletes the entry from the DBM file. Deleting from a `tied` hash or array may not necessarily return anything; it depends on the implementation of the `tied` package's DELETE method, which may do whatever it pleases.

The `delete local EXPR` construct localizes the deletion to the current block at run time. Until the block exits, elements locally deleted temporarily no longer exist. See "Localized deletion of elements of composite types" in perlsub.

```
my %hash = (foo => 11, bar => 22, baz => 33);
my $scalar = delete $hash{foo};         # $scalar is 11
$scalar = delete @hash{qw(foo bar)}; # $scalar is 22
my @array  = delete @hash{qw(foo baz)}; # @array  is (undef,33)
```

The following (inefficiently) deletes all the values of `%HASH` and `@ARRAY`:

```
foreach my $key (keys %HASH) {
    delete $HASH{$key};
}

foreach my $index (0 .. $#ARRAY) {
    delete $ARRAY[$index];
}
```

And so do these:

```
delete @HASH{keys %HASH};

delete @ARRAY[0 .. $#ARRAY];
```

But both are slower than assigning the empty list or undefining `%HASH` or `@ARRAY`, which is the customary way to empty out an aggregate:

```
%HASH = ();     # completely empty %HASH
undef %HASH;    # forget %HASH ever existed

@ARRAY = ();    # completely empty @ARRAY
```

```
undef @ARRAY;   # forget @ARRAY ever existed
```

The EXPR can be arbitrarily complicated provided its final operation is an element or slice of an aggregate:

```
delete $ref->[$x][$y]{$key};
delete @{$ref->[$x][$y]}{$key1, $key2, @morekeys};

delete $ref->[$x][$y][$index];
delete @{$ref->[$x][$y]}[$index1, $index2, @moreindices];
```

die LIST

`die` raises an exception. Inside an `eval` the error message is stuffed into `$@` and the `eval` is terminated with the undefined value. If the exception is outside of all enclosing `eval`s, then the uncaught exception prints LIST to `STDERR` and exits with a non-zero value. If you need to exit the process with a specific exit code, see `exit`.

Equivalent examples:

```
die "Can't cd to spool: $!\n" unless chdir '/usr/spool/news';
chdir '/usr/spool/news' or die "Can't cd to spool: $!\n"
```

If the last element of LIST does not end in a newline, the current script line number and input line number (if any) are also printed, and a newline is supplied. Note that the ''input line number'' (also known as ''chunk'') is subject to whatever notion of ''line'' happens to be currently in effect, and is also available as the special variable `$.`. See ''$/'' in perlvar and ''$.'' in perlvar.

Hint: sometimes appending `", stopped"` to your message will cause it to make better sense when the string `"at foo line 123"` is appended. Suppose you are running script ''canasta''.

```
die "/etc/games is no good";
die "/etc/games is no good, stopped";
```

produce, respectively

```
/etc/games is no good at canasta line 123.
/etc/games is no good, stopped at canasta line 123.
```

If the output is empty and `$@` already contains a value (typically from a previous `eval`) that value is reused after appending `"\t...propagated"`. This is useful for propagating exceptions:

```
eval { ... };
die unless $@ =~ /Expected exception/;
```

If the output is empty and `$@` contains an object reference that has a `PROPAGATE` method, that method will be called with additional file and line number parameters. The return value replaces the value in `$@`; i.e., as if `$@ = eval { $@->PROPAGATE(__FILE__, __LINE__) };` were called.

If `$@` is empty, then the string `"Died"` is used.

If an uncaught exception results in interpreter exit, the exit code is determined from the values of `$!` and `$?` with this pseudocode:

```
exit $! if $!;              # errno
exit $? >> 8 if $? >> 8;    # child exit status
exit 255;                   # last resort
```

As with `exit`, `$?` is set prior to unwinding the call stack; any `DESTROY` or `END` handlers can then alter this value, and thus Perl's exit code.

The intent is to squeeze as much possible information about the likely cause into the limited space of the system exit code. However, as `$!` is the value of C's `errno`, which can be set by any system call, this means that the value of the exit code used by `die` can be non-predictable, so should not be relied upon, other than to be non-zero.

You can also call `die` with a reference argument, and if this is trapped within an `eval`, `$@` contains that reference. This permits more elaborate exception handling using objects that maintain arbitrary

state about the exception. Such a scheme is sometimes preferable to matching particular string values of `$@` with regular expressions. Because `$@` is a global variable and `eval` may be used within object implementations, be careful that analyzing the error object doesn't replace the reference in the global variable. It's easiest to make a local copy of the reference before any manipulations. Here's an example:

```
use Scalar::Util "blessed";

eval { ... ; die Some::Module::Exception->new( FOO => "bar" ) };
if (my $ev_err = $@) {
    if (blessed($ev_err)
        && $ev_err->isa("Some::Module::Exception")) {
        # handle Some::Module::Exception
    }
    else {
        # handle all other possible exceptions
    }
}
```

Because Perl stringifies uncaught exception messages before display, you'll probably want to overload stringification operations on exception objects. See overload for details about that.

You can arrange for a callback to be run just before the `die` does its deed, by setting the `$SIG{__DIE__}` hook. The associated handler is called with the error text and can change the error message, if it sees fit, by calling `die` again. See "%SIG" in perlvar for details on setting `%SIG` entries, and `eval` for some examples. Although this feature was to be run only right before your program was to exit, this is not currently so: the `$SIG{__DIE__}` hook is currently called even inside `eval`ed blocks/strings! If one wants the hook to do nothing in such situations, put

```
die @_ if $^S;
```

as the first line of the handler (see "$^S" in perlvar). Because this promotes strange action at a distance, this counterintuitive behavior may be fixed in a future release.

See also `exit`, `warn`, and the Carp module.

do BLOCK

Not really a function. Returns the value of the last command in the sequence of commands indicated by BLOCK. When modified by the `while` or `until` loop modifier, executes the BLOCK once before testing the loop condition. (On other statements the loop modifiers test the conditional first.)

`do BLOCK` does *not* count as a loop, so the loop control statements `next`, `last`, or `redo` cannot be used to leave or restart the block. See perlsyn for alternative strategies.

do EXPR

Uses the value of EXPR as a filename and executes the contents of the file as a Perl script:

```
# load the exact specified file (./ and ../ special-cased)
do '/foo/stat.pl';
do './stat.pl';
do '../foo/stat.pl';

# search for the named file within @INC
do 'stat.pl';
do 'foo/stat.pl';
```

`do './stat.pl'` is largely like

```
eval `cat stat.pl`;
```

except that it's more concise, runs no external processes, and keeps track of the current filename for error messages. It also differs in that code evaluated with `do FILE` cannot see lexicals in the enclosing scope; `eval STRING` does. It's the same, however, in that it does reparse the file every time you call it, so you probably don't want to do this inside a loop.

Using `do` with a relative path (except for *./* and *../*), like

```
do 'foo/stat.pl';
```

will search the `@INC` directories, and update `%INC` if the file is found. See "@INC" in perlvar and "%INC" in perlvar for these variables. In particular, note that whilst historically `@INC` contained '.' (the current directory) making these two cases equivalent, that is no longer necessarily the case, as '.' is not included in `@INC` by default in perl versions 5.26.0 onwards. Instead, perl will now warn:

```
do "stat.pl" failed, '.' is no longer in @INC;
did you mean do "./stat.pl"?
```

If `do` can read the file but cannot compile it, it returns `undef` and sets an error message in `$@`. If `do` cannot read the file, it returns undef and sets `$!` to the error. Always check `$@` first, as compilation could fail in a way that also sets `$!`. If the file is successfully compiled, `do` returns the value of the last expression evaluated.

Inclusion of library modules is better done with the `use` and `require` operators, which also do automatic error checking and raise an exception if there's a problem.

You might like to use `do` to read in a program configuration file. Manual error checking can be done this way:

```
# Read in config files: system first, then user.
# Beware of using relative pathnames here.
for $file ("/share/prog/defaults.rc",
           "$ENV{HOME}/.someprogrc")
{
    unless ($return = do $file) {
        warn "couldn't parse $file: $@" if $@;
        warn "couldn't do $file: $!"    unless defined $return;
        warn "couldn't run $file"       unless $return;
    }
}
```

dump LABEL
dump EXPR
dump

This function causes an immediate core dump. See also the **-u** command-line switch in perlrun, which does the same thing. Primarily this is so that you can use the **undump** program (not supplied) to turn your core dump into an executable binary after having initialized all your variables at the beginning of the program. When the new binary is executed it will begin by executing a `goto LABEL` (with all the restrictions that `goto` suffers). Think of it as a goto with an intervening core dump and reincarnation. If `LABEL` is omitted, restarts the program from the top. The `dump EXPR` form, available starting in Perl 5.18.0, allows a name to be computed at run time, being otherwise identical to `dump LABEL`.

WARNING: Any files opened at the time of the dump will *not* be open any more when the program is reincarnated, with possible resulting confusion by Perl.

This function is now largely obsolete, mostly because it's very hard to convert a core file into an executable. That's why you should now invoke it as `CORE::dump()` if you don't want to be warned against a possible typo.

Unlike most named operators, this has the same precedence as assignment. It is also exempt from the looks-like-a-function rule, so `dump ("foo")."bar"` will cause "bar" to be part of the argument to `dump`.

Portability issues: "dump" in perlport.

each HASH
each ARRAY

When called on a hash in list context, returns a 2-element list consisting of the key and value for the next element of a hash. In Perl 5.12 and later only, it will also return the index and value for the next element of an array so that you can iterate over it; older Perls consider this a syntax error. When called in scalar context, returns only the key (not the value) in a hash, or the index in an array.

Hash entries are returned in an apparently random order. The actual random order is specific to a given hash; the exact same series of operations on two hashes may result in a different order for each hash. Any insertion into the hash may change the order, as will any deletion, with the exception that the most recent key returned by `each` or `keys` may be deleted without changing the order. So long as a given hash is unmodified you may rely on `keys`, `values` and `each` to repeatedly return the same order as each other. See "Algorithmic Complexity Attacks" in perlsec for details on why hash order is randomized. Aside from the guarantees provided here the exact details of Perl's hash algorithm and the hash traversal order are subject to change in any release of Perl.

After `each` has returned all entries from the hash or array, the next call to `each` returns the empty list in list context and `undef` in scalar context; the next call following *that* one restarts iteration. Each hash or array has its own internal iterator, accessed by `each`, `keys`, and `values`. The iterator is implicitly reset when `each` has reached the end as just described; it can be explicitly reset by calling `keys` or `values` on the hash or array. If you add or delete a hash's elements while iterating over it, the effect on the iterator is unspecified; for example, entries may be skipped or duplicated—so don't do that. Exception: It is always safe to delete the item most recently returned by `each`, so the following code works properly:

```
while (my ($key, $value) = each %hash) {
    print $key, "\n";
    delete $hash{$key};   # This is safe
}
```

Tied hashes may have a different ordering behaviour to perl's hash implementation.

This prints out your environment like the *printenv*(1) program, but in a different order:

```
while (my ($key,$value) = each %ENV) {
    print "$key=$value\n";
}
```

Starting with Perl 5.14, an experimental feature allowed `each` to take a scalar expression. This experiment has been deemed unsuccessful, and was removed as of Perl 5.24.

As of Perl 5.18 you can use a bare `each` in a `while` loop, which will set `$_` on every iteration.

```
while (each %ENV) {
    print "$_=$ENV{$_}\n";
}
```

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious syntax errors, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
use 5.012;  # so keys/values/each work on arrays
use 5.018;  # so each assigns to $_ in a lone while test
```

See also `keys`, `values`, and `sort`.

eof FILEHANDLE
eof ()
eof Returns 1 if the next read on FILEHANDLE will return end of file *or* if FILEHANDLE is not open. FILEHANDLE may be an expression whose value gives the real filehandle. (Note that this function actually reads a character and then `ungetc`s it, so isn't useful in an interactive context.) Do not read from a terminal file (or call `eof(FILEHANDLE)` on it) after end-of-file is reached. File types such as terminals may lose the end-of-file condition if you do.

An `eof` without an argument uses the last file read. Using `eof()` with empty parentheses is different. It refers to the pseudo file formed from the files listed on the command line and accessed via the <> operator. Since <> isn't explicitly opened, as a normal filehandle is, an `eof()` before <> has been used will cause `@ARGV` to be examined to determine if input is available. Similarly, an `eof()` after <> has returned end-of-file will assume you are processing another `@ARGV` list, and if you haven't set `@ARGV`, will read input from `STDIN`; see "I/O Operators" in perlop.

In a `while (<>)` loop, `eof` or `eof(ARGV)` can be used to detect the end of each file, whereas `eof()` will detect the end of the very last file only. Examples:

```
# reset line numbering on each input file
while (<>) {
    next if /^\s*#/;  # skip comments
    print "$.\t$_";
} continue {
    close ARGV if eof;  # Not eof()!
}

# insert dashes just before last line of last file
while (<>) {
    if (eof()) {  # check for end of last file
        print "--------------\n";
    }
    print;
    last if eof();     # needed if we're reading from a terminal
}
```

Practical hint: you almost never need to use `eof` in Perl, because the input operators typically return `undef` when they run out of data or encounter an error.

eval EXPR
eval BLOCK
eval

`eval` in all its forms is used to execute a little Perl program, trapping any errors encountered so they don't crash the calling program.

Plain `eval` with no argument is just `eval EXPR`, where the expression is understood to be contained in `$_`. Thus there are only two real `eval` forms; the one with an EXPR is often called ''string eval''. In a string eval, the value of the expression (which is itself determined within scalar context) is first parsed, and if there were no errors, executed as a block within the lexical context of the current Perl program. This form is typically used to delay parsing and subsequent execution of the text of EXPR until run time. Note that the value is parsed every time the `eval` executes.

The other form is called ''block eval''. It is less general than string eval, but the code within the BLOCK is parsed only once (at the same time the code surrounding the `eval` itself was parsed) and executed within the context of the current Perl program. This form is typically used to trap exceptions more efficiently than the first, while also providing the benefit of checking the code within BLOCK at compile time. BLOCK is parsed and compiled just once. Since errors are trapped, it often is used to check if a given feature is available.

In both forms, the value returned is the value of the last expression evaluated inside the mini-program; a return statement may also be used, just as with subroutines. The expression providing the return value is evaluated in void, scalar, or list context, depending on the context of the `eval` itself. See `wantarray` for more on how the evaluation context can be determined.

If there is a syntax error or runtime error, or a `die` statement is executed, `eval` returns `undef` in scalar context, or an empty list in list context, and `$@` is set to the error message. (Prior to 5.16, a bug caused `undef` to be returned in list context for syntax errors, but not for runtime errors.) If there was no error, `$@` is set to the empty string. A control flow operator like `last` or `goto` can bypass the setting of `$@`. Beware that using `eval` neither silences Perl from printing warnings to STDERR, nor does it stuff the text of warning messages into `$@`. To do either of those, you have to use the `$SIG{__WARN__}` facility, or turn off warnings inside the BLOCK or EXPR using `no warnings 'all'`. See `warn`, perlvar, and warnings.

Note that, because `eval` traps otherwise-fatal errors, it is useful for determining whether a particular feature (such as `socket` or `symlink`) is implemented. It is also Perl's exception-trapping mechanism, where the `die` operator is used to raise exceptions.

Before Perl 5.14, the assignment to `$@` occurred before restoration of localized variables, which means that for your code to run on older versions, a temporary is required if you want to mask some, but not all errors:

```
# alter $@ on nefarious repugnancy only
{
   my $e;
   {
     local $@; # protect existing $@
     eval { test_repugnancy() };
     # $@ =~ /nefarious/ and die $@; # Perl 5.14 and higher only
     $@ =~ /nefarious/ and $e = $@;
   }
   die $e if defined $e
}
```

There are some different considerations for each form:

String eval

Since the return value of EXPR is executed as a block within the lexical context of the current Perl program, any outer lexical variables are visible to it, and any package variable settings or subroutine and format definitions remain afterwards.

Under the ``unicode_eval'' feature

If this feature is enabled (which is the default under a `use 5.16` or higher declaration), EXPR is considered to be in the same encoding as the surrounding program. Thus if `use utf8` is in effect, the string will be treated as being UTF-8 encoded. Otherwise, the string is considered to be a sequence of independent bytes. Bytes that correspond to ASCII-range code points will have their normal meanings for operators in the string. The treatment of the other bytes depends on if the `'unicode_strings"` feature is in effect.

In a plain `eval` without an EXPR argument, being in `use utf8` or not is irrelevant; the UTF-8ness of `$_` itself determines the behavior.

Any `use utf8` or `no utf8` declarations within the string have no effect, and source filters are forbidden. (`unicode_strings`, however, can appear within the string.) See also the `evalbytes` operator, which works properly with source filters.

Variables defined outside the `eval` and used inside it retain their original UTF-8ness. Everything inside the string follows the normal rules for a Perl program with the given state of `use utf8`.

Outside the ``unicode_eval'' feature

In this case, the behavior is problematic and is not so easily described. Here are two bugs that cannot easily be fixed without breaking existing programs:

- It can lose track of whether something should be encoded as UTF-8 or not.
- Source filters activated within `eval` leak out into whichever file scope is currently being compiled. To give an example with the CPAN module Semi::Semicolons:

```
BEGIN { eval "use Semi::Semicolons; # not filtered" }
# filtered here!
```

`evalbytes` fixes that to work the way one would expect:

```
use feature "evalbytes";
BEGIN { evalbytes "use Semi::Semicolons; # filtered" }
# not filtered
```

Problems can arise if the string expands a scalar containing a floating point number. That scalar can expand to letters, such as `"NaN"` or `"Infinity"`; or, within the scope of a `use locale`, the decimal point character may be something other than a dot (such as a comma). None of these are likely to parse as you are likely expecting.

You should be especially careful to remember what's being looked at when:

```
eval $x;        # CASE 1
eval "$x";      # CASE 2

eval '$x';      # CASE 3
```

```
eval { $x };    # CASE 4

eval "\$$x++";  # CASE 5
$$x++;          # CASE 6
```

Cases 1 and 2 above behave identically: they run the code contained in the variable `$x`. (Although case 2 has misleading double quotes making the reader wonder what else might be happening (nothing is).) Cases 3 and 4 likewise behave in the same way: they run the code `'$x'`, which does nothing but return the value of `$x`. (Case 4 is preferred for purely visual reasons, but it also has the advantage of compiling at compile-time instead of at run-time.) Case 5 is a place where normally you *would* like to use double quotes, except that in this particular situation, you can just use symbolic references instead, as in case 6.

An `eval ''` executed within a subroutine defined in the `DB` package doesn't see the usual surrounding lexical scope, but rather the scope of the first non-DB piece of code that called it. You don't normally need to worry about this unless you are writing a Perl debugger.

The final semicolon, if any, may be omitted from the value of EXPR.

Block eval

If the code to be executed doesn't vary, you may use the eval-BLOCK form to trap run-time errors without incurring the penalty of recompiling each time. The error, if any, is still returned in `$@`. Examples:

```
# make divide-by-zero nonfatal
eval { $answer = $a / $b; }; warn $@ if $@;

# same thing, but less efficient
eval '$answer = $a / $b'; warn $@ if $@;

# a compile-time error
eval { $answer = }; # WRONG

# a run-time error
eval '$answer =';   # sets $@
```

If you want to trap errors when loading an XS module, some problems with the binary interface (such as Perl version skew) may be fatal even with `eval` unless `$ENV{PERL_DL_NONLAZY}` is set. See perlrun.

Using the `eval {}` form as an exception trap in libraries does have some issues. Due to the current arguably broken state of `__DIE__` hooks, you may wish not to trigger any `__DIE__` hooks that user code may have installed. You can use the `local $SIG{__DIE__}` construct for this purpose, as this example shows:

```
# a private exception trap for divide-by-zero
eval { local $SIG{'__DIE__'}; $answer = $a / $b; };
warn $@ if $@;
```

This is especially significant, given that `__DIE__` hooks can call `die` again, which has the effect of changing their error messages:

```
# __DIE__ hooks may modify error messages
{
   local $SIG{'__DIE__'} =
          sub { (my $x = $_[0]) =~ s/foo/bar/g; die $x };
   eval { die "foo lives here" };
   print $@ if $@;                # prints "bar lives here"
}
```

Because this promotes action at a distance, this counterintuitive behavior may be fixed in a future release.

`eval BLOCK` does *not* count as a loop, so the loop control statements `next`, `last`, or `redo` cannot be used to leave or restart the block.

The final semicolon, if any, may be omitted from within the BLOCK.

evalbytes EXPR
evalbytes

This function is similar to a string eval, except it always parses its argument (or `$_` if EXPR is omitted) as a string of independent bytes.

If called when `use utf8` is in effect, the string will be assumed to be encoded in UTF-8, and `evalbytes` will make a temporary copy to work from, downgraded to non-UTF-8. If this is not possible (because one or more characters in it require UTF-8), the `evalbytes` will fail with the error stored in `$@`.

Bytes that correspond to ASCII-range code points will have their normal meanings for operators in the string. The treatment of the other bytes depends on if the `'unicode_strings"` feature is in effect.

Of course, variables that are UTF-8 and are referred to in the string retain that:

```
my $a = "\x{100}";
evalbytes 'print ord $a, "\n"';
```

prints

```
256
```

and `$@` is empty.

Source filters activated within the evaluated code apply to the code itself.

`evalbytes` is available starting in Perl v5.16. To access it, you must say `CORE::evalbytes`, but you can omit the `CORE::` if the `"evalbytes"` feature is enabled. This is enabled automatically with a `use v5.16` (or higher) declaration in the current scope.

exec LIST
exec PROGRAM LIST

The `exec` function executes a system command *and never returns*; use `system` instead of `exec` if you want it to return. It fails and returns false only if the command does not exist *and* it is executed directly instead of via your system's command shell (see below).

Since it's a common mistake to use `exec` instead of `system`, Perl warns you if `exec` is called in void context and if there is a following statement that isn't `die`, `warn`, or `exit` (if warnings are enabled — but you always do that, right?). If you *really* want to follow an `exec` with some other statement, you can use one of these styles to avoid the warning:

```
exec ('foo')   or print STDERR "couldn't exec foo: $!";
{ exec ('foo') }; print STDERR "couldn't exec foo: $!";
```

If there is more than one argument in LIST, this calls *execvp*(3) with the arguments in LIST. If there is only one element in LIST, the argument is checked for shell metacharacters, and if there are any, the entire argument is passed to the system's command shell for parsing (this is `/bin/sh -c` on Unix platforms, but varies on other platforms). If there are no shell metacharacters in the argument, it is split into words and passed directly to `execvp`, which is more efficient. Examples:

```
exec '/bin/echo', 'Your arguments are: ', @ARGV;
exec "sort $outfile | uniq";
```

If you don't really want to execute the first argument, but want to lie to the program you are executing about its own name, you can specify the program you actually want to run as an "indirect object" (without a comma) in front of the LIST, as in `exec PROGRAM LIST`. (This always forces interpretation of the LIST as a multivalued list, even if there is only a single scalar in the list.) Example:

```
my $shell = '/bin/csh';
exec $shell '-sh';    # pretend it's a login shell
```

or, more directly,

```
exec {'/bin/csh'} '-sh';  # pretend it's a login shell
```

When the arguments get executed via the system shell, results are subject to its quirks and capabilities.

See "'STRING'" in perlop for details.

Using an indirect object with `exec` or `system` is also more secure. This usage (which also works fine with `system`) forces interpretation of the arguments as a multivalued list, even if the list had just one argument. That way you're safe from the shell expanding wildcards or splitting up words with whitespace in them.

```
my @args = ( "echo surprise" );

exec @args;               # subject to shell escapes
                            # if @args == 1
exec { $args[0] } @args;  # safe even with one-arg list
```

The first version, the one without the indirect object, ran the *echo* program, passing it `"surprise"` an argument. The second version didn't; it tried to run a program named *"echo surprise"*, didn't find it, and set `$?` to a non-zero value indicating failure.

On Windows, only the `exec PROGRAM LIST` indirect object syntax will reliably avoid using the shell; `exec LIST`, even with more than one element, will fall back to the shell if the first spawn fails.

Perl attempts to flush all files opened for output before the exec, but this may not be supported on some platforms (see perlport). To be safe, you may need to set `$|` (`$AUTOFLUSH` in English) or call the `autoflush` method of `IO::Handle` on any open handles to avoid lost output.

Note that `exec` will not call your `END` blocks, nor will it invoke `DESTROY` methods on your objects.

Portability issues: "exec" in perlport.

exists EXPR

Given an expression that specifies an element of a hash, returns true if the specified element in the hash has ever been initialized, even if the corresponding value is undefined.

```
print "Exists\n"    if exists $hash{$key};
print "Defined\n"   if defined $hash{$key};
print "True\n"      if $hash{$key};
```

exists may also be called on array elements, but its behavior is much less obvious and is strongly tied to the use of `delete` on arrays.

WARNING: Calling `exists` on array values is strongly discouraged. The notion of deleting or checking the existence of Perl array elements is not conceptually coherent, and can lead to surprising behavior.

```
print "Exists\n"    if exists $array[$index];
print "Defined\n"   if defined $array[$index];
print "True\n"      if $array[$index];
```

A hash or array element can be true only if it's defined and defined only if it exists, but the reverse doesn't necessarily hold true.

Given an expression that specifies the name of a subroutine, returns true if the specified subroutine has ever been declared, even if it is undefined. Mentioning a subroutine name for exists or defined does not count as declaring it. Note that a subroutine that does not exist may still be callable: its package may have an `AUTOLOAD` method that makes it spring into existence the first time that it is called; see perlsub.

```
print "Exists\n"  if exists &subroutine;
print "Defined\n" if defined &subroutine;
```

Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash or array key lookup or subroutine name:

```
if (exists $ref->{A}->{B}->{$key})  { }
if (exists $hash{A}{B}{$key})       { }

if (exists $ref->{A}->{B}->[$ix])   { }
if (exists $hash{A}{B}[$ix])        { }
```

```
if (exists &{$ref->{A}{B}{$key}})    { }
```

Although the most deeply nested array or hash element will not spring into existence just because its existence was tested, any intervening ones will. Thus `$ref->{"A"}` and `$ref->{"A"}->{"B"}` will spring into existence due to the existence test for the `$key` element above. This happens anywhere the arrow operator is used, including even here:

```
undef $ref;
if (exists $ref->{"Some key"})    { }
print $ref;  # prints HASH(0x80d3d5c)
```

This surprising autovivification in what does not at first — or even second — glance appear to be an lvalue context may be fixed in a future release.

Use of a subroutine call, rather than a subroutine name, as an argument to `exists` is an error.

```
exists &sub;    # OK
exists &sub();  # Error
```

exit EXPR

exit Evaluates EXPR and exits immediately with that value. Example:

```
my $ans = <STDIN>;
exit 0 if $ans =~ /^[Xx]/;
```

See also `die`. If EXPR is omitted, exits with `0` status. The only universally recognized values for EXPR are `0` for success and `1` for error; other values are subject to interpretation depending on the environment in which the Perl program is running. For example, exiting 69 (EX_UNAVAILABLE) from a *sendmail* incoming-mail filter will cause the mailer to return the item undelivered, but that's not true everywhere.

Don't use `exit` to abort a subroutine if there's any chance that someone might want to trap whatever error happened. Use `die` instead, which can be trapped by an `eval`.

The `exit` function does not always exit immediately. It calls any defined `END` routines first, but these `END` routines may not themselves abort the exit. Likewise any object destructors that need to be called are called before the real exit. `END` routines and destructors can change the exit status by modifying `$?`. If this is a problem, you can call `POSIX::_exit($status)` to avoid `END` and destructor processing. See perlmod for details.

Portability issues: "exit" in perlport.

exp EXPR

exp Returns *e* (the natural logarithm base) to the power of EXPR. If EXPR is omitted, gives `exp($_)`.

fc EXPR

fc Returns the casefolded version of EXPR. This is the internal function implementing the `\F` escape in double-quoted strings.

Casefolding is the process of mapping strings to a form where case differences are erased; comparing two strings in their casefolded form is effectively a way of asking if two strings are equal, regardless of case.

Roughly, if you ever found yourself writing this

```
lc($this) eq lc($that)    # Wrong!
    # or
uc($this) eq uc($that)    # Also wrong!
    # or
$this =~ /^\Q$that\E\z/i  # Right!
```

Now you can write

```
fc($this) eq fc($that)
```

And get the correct results.

Perl only implements the full form of casefolding, but you can access the simple folds using

"***casefold()***" in Unicode::UCD and "***prop_invmap()***" in Unicode::UCD. For further information on casefolding, refer to the Unicode Standard, specifically sections 3.13 `Default Case Operations`, 4.2 `Case-Normative`, and 5.18 `Case Mappings`, available at <http://www.unicode.org/versions/latest/>, as well as the Case Charts available at <http://www.unicode.org/charts/case/>.

If EXPR is omitted, uses `$_`.

This function behaves the same way under various pragmas, such as within `"use feature 'unicode_strings"`, as `lc` does, with the single exception of `fc` of *LATIN CAPITAL LETTER SHARP S* (U+1E9E) within the scope of `use locale`. The foldcase of this character would normally be `"ss"`, but as explained in the `lc` section, case changes that cross the 255/256 boundary are problematic under locales, and are hence prohibited. Therefore, this function under locale returns instead the string `"\x{17F}\x{17F}"`, which is the *LATIN SMALL LETTER LONG S*. Since that character itself folds to `"s"`, the string of two of them together should be equivalent to a single U+1E9E when foldcased.

While the Unicode Standard defines two additional forms of casefolding, one for Turkic languages and one that never maps one character into multiple characters, these are not provided by the Perl core. However, the CPAN module `Unicode::Casing` may be used to provide an implementation.

`fc` is available only if the `"fc"` feature is enabled or if it is prefixed with `CORE::`. The `"fc"` feature is enabled automatically with a `use v5.16` (or higher) declaration in the current scope.

fcntl FILEHANDLE,FUNCTION,SCALAR

Implements the *fcntl*(2) function. You'll probably have to say

```
use Fcntl;
```

first to get the correct constant definitions. Argument processing and value returned work just like `ioctl` below. For example:

```
use Fcntl;
my $flags = fcntl($filehandle, F_GETFL, 0)
    or die "Can't fcntl F_GETFL: $!";
```

You don't have to check for `defined` on the return from `fcntl`. Like `ioctl`, it maps a `0` return from the system call into `"0 but true"` in Perl. This string is true in boolean context and `0` in numeric context. It is also exempt from the normal `Argument "..." isn't numeric` warnings on improper numeric conversions.

Note that `fcntl` raises an exception if used on a machine that doesn't implement *fcntl*(2). See the Fcntl module or your *fcntl*(2) manpage to learn what functions are available on your system.

Here's an example of setting a filehandle named `$REMOTE` to be non-blocking at the system level. You'll have to negotiate `$|` on your own, though.

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

my $flags = fcntl($REMOTE, F_GETFL, 0)
    or die "Can't get flags for the socket: $!\n";

fcntl($REMOTE, F_SETFL, $flags | O_NONBLOCK)
    or die "Can't set flags for the socket: $!\n";
```

Portability issues: "fcntl" in perlport.

__FILE__

A special token that returns the name of the file in which it occurs.

fileno FILEHANDLE

Returns the file descriptor for a filehandle, or undefined if the filehandle is not open. If there is no real file descriptor at the OS level, as can happen with filehandles connected to memory objects via `open` with a reference for the third argument, -1 is returned.

This is mainly useful for constructing bitmaps for `select` and low-level POSIX tty-handling operations. If FILEHANDLE is an expression, the value is taken as an indirect filehandle, generally its

name.

You can use this to find out whether two handles refer to the same underlying descriptor:

```
if (fileno($this) != -1 && fileno($this) == fileno($that)) {
    print "\$this and \$that are dups\n";
} elsif (fileno($this) != -1 && fileno($that) != -1) {
    print "\$this and \$that have different " .
        "underlying file descriptors\n";
} else {
    print "At least one of \$this and \$that does " .
        "not have a real file descriptor\n";
}
```

The behavior of `fileno` on a directory handle depends on the operating system. On a system with *dirfd*(3) or similar, `fileno` on a directory handle returns the underlying file descriptor associated with the handle; on systems with no such support, it returns the undefined value, and sets `$!` (errno).

flock FILEHANDLE,OPERATION

Calls *flock*(2), or an emulation of it, on FILEHANDLE. Returns true for success, false on failure. Produces a fatal error if used on a machine that doesn't implement *flock*(2), *fcntl*(2) locking, or *lockf*(3). `flock` is Perl's portable file-locking interface, although it locks entire files only, not records.

Two potentially non-obvious but traditional `flock` semantics are that it waits indefinitely until the lock is granted, and that its locks are **merely advisory**. Such discretionary locks are more flexible, but offer fewer guarantees. This means that programs that do not also use `flock` may modify files locked with `flock`. See perlport, your port's specific documentation, and your system-specific local manpages for details. It's best to assume traditional behavior if you're writing portable programs. (But if you're not, you should as always feel perfectly free to write for your own system's idiosyncrasies (sometimes called "features"). Slavish adherence to portability concerns shouldn't get in the way of your getting your job done.)

OPERATION is one of LOCK_SH, LOCK_EX, or LOCK_UN, possibly combined with LOCK_NB. These constants are traditionally valued 1, 2, 8 and 4, but you can use the symbolic names if you import them from the Fcntl module, either individually, or as a group using the `:flock` tag. LOCK_SH requests a shared lock, LOCK_EX requests an exclusive lock, and LOCK_UN releases a previously requested lock. If LOCK_NB is bitwise-or'ed with LOCK_SH or LOCK_EX, then `flock` returns immediately rather than blocking waiting for the lock; check the return status to see if you got it.

To avoid the possibility of miscoordination, Perl now flushes FILEHANDLE before locking or unlocking it.

Note that the emulation built with *lockf*(3) doesn't provide shared locks, and it requires that FILEHANDLE be open with write intent. These are the semantics that *lockf*(3) implements. Most if not all systems implement *lockf*(3) in terms of *fcntl*(2) locking, though, so the differing semantics shouldn't bite too many people.

Note that the *fcntl*(2) emulation of *flock*(3) requires that FILEHANDLE be open with read intent to use LOCK_SH and requires that it be open with write intent to use LOCK_EX.

Note also that some versions of `flock` cannot lock things over the network; you would need to use the more system-specific `fcntl` for that. If you like you can force Perl to ignore your system's *flock*(2) function, and so provide its own *fcntl*(2)–based emulation, by passing the switch `-Ud_flock` to the *Configure* program when you configure and build a new Perl.

Here's a mailbox appender for BSD systems.

```
# import LOCK_* and SEEK_END constants
use Fcntl qw(:flock SEEK_END);

sub lock {
    my ($fh) = @_;
    flock($fh, LOCK_EX) or die "Cannot lock mailbox - $!\n";
```

```
    # and, in case someone appended while we were waiting...
    seek($fh, 0, SEEK_END) or die "Cannot seek - $!\n";
}

sub unlock {
    my ($fh) = @_;
    flock($fh, LOCK_UN) or die "Cannot unlock mailbox - $!\n";
}

open(my $mbox, ">>", "/usr/spool/mail/$ENV{'USER'}")
    or die "Can't open mailbox: $!";

lock($mbox);
print $mbox $msg,"\n\n";
unlock($mbox);
```

On systems that support a real *flock*(2), locks are inherited across `fork` calls, whereas those that must resort to the more capricious *fcntl*(2) function lose their locks, making it seriously harder to write servers.

See also DB_File for other `flock` examples.

Portability issues: "flock" in perlport.

fork

Does a *fork*(2) system call to create a new process running the same program at the same point. It returns the child pid to the parent process, `0` to the child process, or `undef` if the fork is unsuccessful. File descriptors (and sometimes locks on those descriptors) are shared, while everything else is copied. On most systems supporting *fork*(2), great care has gone into making it extremely efficient (for example, using copy-on-write technology on data pages), making it the dominant paradigm for multitasking over the last few decades.

Perl attempts to flush all files opened for output before forking the child process, but this may not be supported on some platforms (see perlport). To be safe, you may need to set `$|` (`$AUTOFLUSH` in English) or call the `autoflush` method of `IO::Handle` on any open handles to avoid duplicate output.

If you `fork` without ever waiting on your children, you will accumulate zombies. On some systems, you can avoid this by setting `$SIG{CHLD}` to `"IGNORE"`. See also perlipc for more examples of forking and reaping moribund children.

Note that if your forked child inherits system file descriptors like STDIN and STDOUT that are actually connected by a pipe or socket, even if you exit, then the remote server (such as, say, a CGI script or a backgrounded job launched from a remote shell) won't think you're done. You should reopen those to */dev/null* if it's any issue.

On some platforms such as Windows, where the *fork*(2) system call is not available, Perl can be built to emulate `fork` in the Perl interpreter. The emulation is designed, at the level of the Perl program, to be as compatible as possible with the "Unix" *fork*(2). However it has limitations that have to be considered in code intended to be portable. See perlfork for more details.

Portability issues: "fork" in perlport.

format

Declare a picture format for use by the `write` function. For example:

```
format Something =
    Test: @<<<<<<<< @||||| @>>>>>
          $str,     $%,    '$' . int($num)
.

$str = "widget";
$num = $cost/$quantity;
$~ = 'Something';
```

```
write;
```

See perlform for many details and examples.

formline PICTURE,LIST

This is an internal function used by `formats`, though you may call it, too. It formats (see perlform) a list of values according to the contents of PICTURE, placing the output into the format output accumulator, `$^A` (or `$ACCUMULATOR` in English). Eventually, when a `write` is done, the contents of `$^A` are written to some filehandle. You could also read `$^A` and then set `$^A` back to `""`. Note that a format typically does one `formline` per line of form, but the `formline` function itself doesn't care how many newlines are embedded in the PICTURE. This means that the `~` and `~~` tokens treat the entire PICTURE as a single line. You may therefore need to use multiple formlines to implement a single record format, just like the `format` compiler.

Be careful if you put double quotes around the picture, because an `@` character may be taken to mean the beginning of an array name. `formline` always returns true. See perlform for other examples.

If you are trying to use this instead of `write` to capture the output, you may find it easier to open a filehandle to a scalar (`open my $fh, ">", \$output`) and write to that instead.

getc FILEHANDLE

getc

Returns the next character from the input file attached to FILEHANDLE, or the undefined value at end of file or if there was an error (in the latter case `$!` is set). If FILEHANDLE is omitted, reads from STDIN. This is not particularly efficient. However, it cannot be used by itself to fetch single characters without waiting for the user to hit enter. For that, try something more like:

```
if ($BSD_STYLE) {
    system "stty cbreak </dev/tty >/dev/tty 2>&1";
}
else {
    system "stty", '-icanon', 'eol', "\001";
}

my $key = getc(STDIN);

if ($BSD_STYLE) {
    system "stty -cbreak </dev/tty >/dev/tty 2>&1";
}
else {
    system 'stty', 'icanon', 'eol', '^@'; # ASCII NUL
}
print "\n";
```

Determination of whether `$BSD_STYLE` should be set is left as an exercise to the reader.

The `POSIX::getattr` function can do this more portably on systems purporting POSIX compliance. See also the `Term::ReadKey` module on CPAN.

getlogin

This implements the C library function of the same name, which on most systems returns the current login from */etc/utmp*, if any. If it returns the empty string, use `getpwuid`.

```
my $login = getlogin || getpwuid($<) || "Kilroy";
```

Do not consider `getlogin` for authentication: it is not as secure as `getpwuid`.

Portability issues: "getlogin" in perlport.

getpeername SOCKET

Returns the packed sockaddr address of the other end of the SOCKET connection.

```
use Socket;
my $hersockaddr    = getpeername($sock);
my ($port, $iaddr) = sockaddr_in($hersockaddr);
my $herhostname    = gethostbyaddr($iaddr, AF_INET);
my $herstraddr     = inet_ntoa($iaddr);
```

getpgrp PID
: Returns the current process group for the specified PID. Use a PID of `0` to get the current process group for the current process. Will raise an exception if used on a machine that doesn't implement *getpgrp*(2). If PID is omitted, returns the process group of the current process. Note that the POSIX version of `getpgrp` does not accept a PID argument, so only `PID==0` is truly portable.

  Portability issues: "getpgrp" in perlport.

getppid
: Returns the process id of the parent process.

  Note for Linux users: Between v5.8.1 and v5.16.0 Perl would work around non-POSIX thread semantics the minority of Linux systems (and Debian GNU/kFreeBSD systems) that used LinuxThreads, this emulation has since been removed. See the documentation for $$ for details.

  Portability issues: "getppid" in perlport.

getpriority WHICH,WHO
: Returns the current priority for a process, a process group, or a user. (See *getpriority*(2).) Will raise a fatal exception if used on a machine that doesn't implement *getpriority*(2).

  Portability issues: "getpriority" in perlport.

getpwnam NAME
getgrnam NAME
gethostbyname NAME
getnetbyname NAME
getprotobyname NAME
getpwuid UID
getgrgid GID
getservbyname NAME,PROTO
gethostbyaddr ADDR,ADDRTYPE
getnetbyaddr ADDR,ADDRTYPE
getprotobynumber NUMBER
getservbyport PORT,PROTO
getpwent
getgrent
gethostent
getnetent
getprotoent
getservent
setpwent
setgrent
sethostent STAYOPEN
setnetent STAYOPEN
setprotoent STAYOPEN
setservent STAYOPEN
endpwent
endgrent
endhostent
endnetent
endprotoent
endservent
: These routines are the same as their counterparts in the system C library. In list context, the return values from the various get routines are as follows:

```
#    0        1          2           3          4
my ( $name,   $passwd,   $gid,       $members  ) = getgr*
my ( $name,   $aliases,  $addrtype,  $net      ) = getnet*
my ( $name,   $aliases,  $port,      $proto    ) = getserv*
my ( $name,   $aliases,  $proto                ) = getproto*
my ( $name,   $aliases,  $addrtype,  $length,  @addrs ) = gethost*
my ( $name,   $passwd,   $uid,       $gid,     $quota,
   $comment,  $gcos,     $dir,       $shell,   $expire ) = getpw*
#    5        6          7           8         9
```

(If the entry doesn't exist, the return value is a single meaningless true value.)

The exact meaning of the `$gcos` field varies but it usually contains the real name of the user (as opposed to the login name) and other information pertaining to the user. Beware, however, that in many system users are able to change this information and therefore it cannot be trusted and therefore the `$gcos` is tainted (see perlsec). The `$passwd` and `$shell`, user's encrypted password and login shell, are also tainted, for the same reason.

In scalar context, you get the name, unless the function was a lookup by name, in which case you get the other thing, whatever it is. (If the entry doesn't exist you get the undefined value.) For example:

```
my $uid   = getpwnam($name);
my $name  = getpwuid($num);
my $name  = getpwent();
my $gid   = getgrnam($name);
my $name  = getgrgid($num);
my $name  = getgrent();
# etc.
```

In *getpw*()* the fields `$quota`, `$comment`, and `$expire` are special in that they are unsupported on many systems. If the `$quota` is unsupported, it is an empty scalar. If it is supported, it usually encodes the disk quota. If the `$comment` field is unsupported, it is an empty scalar. If it is supported it usually encodes some administrative comment about the user. In some systems the `$quota` field may be `$change` or `$age`, fields that have to do with password aging. In some systems the `$comment` field may be `$class`. The `$expire` field, if present, encodes the expiration period of the account or the password. For the availability and the exact meaning of these fields in your system, please consult *getpwnam*(3) and your system's *pwd.h* file. You can also find out from within Perl what your `$quota` and `$comment` fields mean and whether you have the `$expire` field by using the `Config` module and the values `d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment`, and `d_pwexpire`. Shadow password files are supported only if your vendor has implemented them in the intuitive fashion that calling the regular C library routines gets the shadow versions if you're running under privilege or if there exists the *shadow*(3) functions as found in System V (this includes Solaris and Linux). Those systems that implement a proprietary shadow password facility are unlikely to be supported.

The `$members` value returned by *getgr*()* is a space-separated list of the login names of the members of the group.

For the *gethost*()* functions, if the `h_errno` variable is supported in C, it will be returned to you via `$?` if the function call fails. The `@addrs` value returned by a successful call is a list of raw addresses returned by the corresponding library call. In the Internet domain, each address is four bytes long; you can unpack it by saying something like:

```
my ($w,$x,$y,$z) = unpack('W4',$addr[0]);
```

The Socket library makes this slightly easier:

```
use Socket;
my $iaddr = inet_aton("127.1"); # or whatever address
my $name  = gethostbyaddr($iaddr, AF_INET);

# or going the other way
my $straddr = inet_ntoa($iaddr);
```

In the opposite way, to resolve a hostname to the IP address you can write this:

```
use Socket;
my $packed_ip = gethostbyname("www.perl.org");
my $ip_address;
if (defined $packed_ip) {
    $ip_address = inet_ntoa($packed_ip);
}
```

Make sure `gethostbyname` is called in SCALAR context and that its return value is checked for definedness.

The `getprotobynumber` function, even though it only takes one argument, has the precedence of a list operator, so beware:

```
getprotobynumber $number eq 'icmp'   # WRONG
getprotobynumber($number eq 'icmp')  # actually means this
getprotobynumber($number) eq 'icmp'  # better this way
```

If you get tired of remembering which element of the return list contains which return value, by-name interfaces are provided in standard modules: `File::stat`, `Net::hostent`, `Net::netent`, `Net::protoent`, `Net::servent`, `Time::gmtime`, `Time::localtime`, and `User::grent`. These override the normal built-ins, supplying versions that return objects with the appropriate names for each field. For example:

```
use File::stat;
use User::pwent;
my $is_his = (stat($filename)->uid == pwent($whoever)->uid);
```

Even though it looks as though they're the same method calls (uid), they aren't, because a `File::stat` object is different from a `User::pwent` object.

Portability issues: "getpwnam" in perlport to "endservent" in perlport.

getsockname SOCKET
: Returns the packed sockaddr address of this end of the SOCKET connection, in case you don't know the address because you have several different IPs that the connection might have come in on.

```
use Socket;
my $mysockaddr = getsockname($sock);
my ($port, $myaddr) = sockaddr_in($mysockaddr);
printf "Connect to %s [%s]\n",
   scalar gethostbyaddr($myaddr, AF_INET),
   inet_ntoa($myaddr);
```

getsockopt SOCKET,LEVEL,OPTNAME
: Queries the option named OPTNAME associated with SOCKET at a given LEVEL. Options may exist at multiple protocol levels depending on the socket type, but at least the uppermost socket level SOL_SOCKET (defined in the `Socket` module) will exist. To query options at another level the protocol number of the appropriate protocol controlling the option should be supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, LEVEL should be set to the protocol number of TCP, which you can get using `getprotobyname`.

  The function returns a packed string representing the requested socket option, or `undef` on error, with the reason for the error placed in `$!`. Just what is in the packed string depends on LEVEL and OPTNAME; consult *getsockopt* (2) for details. A common case is that the option is an integer, in which case the result is a packed integer, which you can decode using `unpack` with the `i` (or `I`) format.

  Here's an example to test whether Nagle's algorithm is enabled on a socket:

```
use Socket qw(:all);

defined(my $tcp = getprotobyname("tcp"))
    or die "Could not determine the protocol number for tcp";
# my $tcp = IPPROTO_TCP; # Alternative
my $packed = getsockopt($socket, $tcp, TCP_NODELAY)
```

```
    or die "getsockopt TCP_NODELAY: $!";
my $nodelay = unpack("I", $packed);
print "Nagle's algorithm is turned ",
       $nodelay ? "off\n" : "on\n";
```

Portability issues: "getsockopt" in perlport.

glob EXPR
glob

In list context, returns a (possibly empty) list of filename expansions on the value of EXPR such as the standard Unix shell */bin/csh* would do. In scalar context, glob iterates through such filename expansions, returning undef when the list is exhausted. This is the internal function implementing the `<*.c>` operator, but you can use it directly. If EXPR is omitted, `$_` is used. The `<*.c>` operator is discussed in more detail in "I/O Operators" in perlop.

Note that `glob` splits its arguments on whitespace and treats each segment as separate pattern. As such, `glob("*.c *.h")` matches all files with a *.c* or *.h* extension. The expression `glob(".* *")` matches all files in the current working directory. If you want to glob filenames that might contain whitespace, you'll have to use extra quotes around the spacey filename to protect it. For example, to glob filenames that have an `e` followed by a space followed by an `f`, use one of:

```
my @spacies = <"*e f*">;
my @spacies = glob '"*e f*"';
my @spacies = glob q("*e f*");
```

If you had to get a variable through, you could do this:

```
my @spacies = glob "'*${var}e f*'";
my @spacies = glob qq("*${var}e f*");
```

If non-empty braces are the only wildcard characters used in the `glob`, no filenames are matched, but potentially many strings are returned. For example, this produces nine strings, one for each pairing of fruits and colors:

```
my @many = glob "{apple,tomato,cherry}={green,yellow,red}";
```

This operator is implemented using the standard `File::Glob` extension. See File::Glob for details, including `bsd_glob`, which does not treat whitespace as a pattern separator.

Portability issues: "glob" in perlport.

gmtime EXPR
gmtime

Works just like `localtime` but the returned values are localized for the standard Greenwich time zone.

Note: When called in list context, `$isdst`, the last value returned by gmtime, is always `0`. There is no Daylight Saving Time in GMT.

Portability issues: "gmtime" in perlport.

goto LABEL
goto EXPR
goto &NAME

The `goto LABEL` form finds the statement labeled with LABEL and resumes execution there. It can't be used to get out of a block or subroutine given to `sort`. It can be used to go almost anywhere else within the dynamic scope, including out of subroutines, but it's usually better to use some other construct such as `last` or `die`. The author of Perl has never felt the need to use this form of `goto` (in Perl, that is; C is another matter). (The difference is that C does not offer named loops combined with loop control. Perl does, and this replaces most structured uses of `goto` in other languages.)

The `goto EXPR` form expects to evaluate `EXPR` to a code reference or a label name. If it evaluates to a code reference, it will be handled like `goto &NAME`, below. This is especially useful for implementing tail recursion via `goto __SUB__`.

If the expression evaluates to a label name, its scope will be resolved dynamically. This allows for computed `goto`s per FORTRAN, but isn't necessarily recommended if you're optimizing for

maintainability:

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

As shown in this example, `goto EXPR` is exempt from the "looks like a function" rule. A pair of parentheses following it does not (necessarily) delimit its argument. `goto("NE")."XT"` is equivalent to `goto NEXT`. Also, unlike most named operators, this has the same precedence as assignment.

Use of `goto LABEL` or `goto EXPR` to jump into a construct is deprecated and will issue a warning. Even then, it may not be used to go into any construct that requires initialization, such as a subroutine or a `foreach` loop. It also can't be used to go into a construct that is optimized away.

The `goto &NAME` form is quite different from the other forms of `goto`. In fact, it isn't a goto in the normal sense at all, and doesn't have the stigma associated with other gotos. Instead, it exits the current subroutine (losing any changes set by `local`) and immediately calls in its place the named subroutine using the current value of `@_`. This is used by `AUTOLOAD` subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place (except that any modifications to `@_` in the current subroutine are propagated to the other subroutine.) After the `goto`, not even `caller` will be able to tell that this routine was called first.

NAME needn't be the name of a subroutine; it can be a scalar variable containing a code reference or a block that evaluates to a code reference.

grep BLOCK LIST
grep EXPR,LIST

This is similar in spirit to, but not the same as, *grep*(1) and its relatives. In particular, it is not limited to using regular expressions.

Evaluates the BLOCK or EXPR for each element of LIST (locally setting `$_` to each element) and returns the list value consisting of those elements for which the expression evaluated to true. In scalar context, returns the number of times the expression was true.

```
my @foo = grep(!/^#/, @bar);    # weed out comments
```

or equivalently,

```
my @foo = grep {!/^#/} @bar;    # weed out comments
```

Note that `$_` is an alias to the list value, so it can be used to modify the elements of the LIST. While this is useful and supported, it can cause bizarre results if the elements of LIST are not variables. Similarly, grep returns aliases into the original list, much as a for loop's index variable aliases the list elements. That is, modifying an element of a list returned by grep (for example, in a `foreach`, `map` or another `grep`) actually modifies the element in the original list. This is usually something to be avoided when writing clear code.

See also `map` for a list composed of the results of the BLOCK or EXPR.

hex EXPR
hex Interprets EXPR as a hex string and returns the corresponding numeric value. If EXPR is omitted, uses `$_`.

```
print hex '0xAf'; # prints '175'
print hex 'aF';   # same
$valid_input =~ /\A(?:0?[xX])?(?:_?[0-9a-fA-F])*\z/
```

A hex string consists of hex digits and an optional `0x` or `x` prefix. Each hex digit may be preceded by a single underscore, which will be ignored. Any other character triggers a warning and causes the rest of the string to be ignored (even leading whitespace, unlike `oct`). Only integers can be represented, and integer overflow triggers a warning.

To convert strings that might start with any of `0`, `0x`, or `0b`, see `oct`. To present something as hex, look into `printf`, `sprintf`, and `unpack`.

import LIST

There is no builtin `import` function. It is just an ordinary method (subroutine) defined (or inherited) by modules that wish to export names to another module. The `use` function calls the `import`

method for the package used. See also `use`, perlmod, and Exporter.

index STR,SUBSTR,POSITION
index STR,SUBSTR
The index function searches for one string within another, but without the wildcard-like behavior of a full regular-expression pattern match. It returns the position of the first occurrence of SUBSTR in STR at or after POSITION. If POSITION is omitted, starts searching from the beginning of the string. POSITION before the beginning of the string or after its end is treated as if it were the beginning or the end, respectively. POSITION and the return value are based at zero. If the substring is not found, `index` returns −1.

int EXPR
int Returns the integer portion of EXPR. If EXPR is omitted, uses `$_`. You should not use this function for rounding: one because it truncates towards `0`, and two because machine representations of floating-point numbers can sometimes produce counterintuitive results. For example, `int(-6.725/0.025)` produces −268 rather than the correct −269; that's because it's really more like −268.99999999999994315658 instead. Usually, the `sprintf`, `printf`, or the `POSIX::floor` and `POSIX::ceil` functions will serve you better than will `int`.

ioctl FILEHANDLE,FUNCTION,SCALAR
Implements the *ioctl*(2) function. You'll probably first have to say

```
require "sys/ioctl.ph";  # probably in
                         # $Config{archlib}/sys/ioctl.ph
```

to get the correct function definitions. If *sys/ioctl.ph* doesn't exist or doesn't have the correct definitions you'll have to roll your own, based on your C header files such as *<sys/ioctl.h>*. (There is a Perl script called **h2ph** that comes with the Perl kit that may help you in this, but it's nontrivial.) SCALAR will be read and/or written depending on the FUNCTION; a C pointer to the string value of SCALAR will be passed as the third argument of the actual `ioctl` call. (If SCALAR has no string value but does have a numeric value, that value will be passed rather than a pointer to the string value. To guarantee this to be true, add a `0` to the scalar before using it.) The `pack` and `unpack` functions may be needed to manipulate the values of structures used by `ioctl`.

The return value of `ioctl` (and `fcntl`) is as follows:

```
if OS returns:      then Perl returns:
    -1               undefined value
     0              string "0 but true"
anything else           that number
```

Thus Perl returns true on success and false on failure, yet you can still easily determine the actual value returned by the operating system:

```
my $retval = ioctl(...) || -1;
printf "System returned %d\n", $retval;
```

The special string `"0 but true"` is exempt from `Argument "..." isn't numeric` warnings on improper numeric conversions.

Portability issues: "ioctl" in perlport.

join EXPR,LIST
Joins the separate strings of LIST into a single string with fields separated by the value of EXPR, and returns that new string. Example:

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

Beware that unlike `split`, `join` doesn't take a pattern as its first argument. Compare `split`.

keys HASH
keys ARRAY
Called in list context, returns a list consisting of all the keys of the named hash, or in Perl 5.12 or later only, the indices of an array. Perl releases prior to 5.12 will produce a syntax error if you try to use an array argument. In scalar context, returns the number of keys or indices.

Hash entries are returned in an apparently random order. The actual random order is specific to a

given hash; the exact same series of operations on two hashes may result in a different order for each hash. Any insertion into the hash may change the order, as will any deletion, with the exception that the most recent key returned by `each` or `keys` may be deleted without changing the order. So long as a given hash is unmodified you may rely on `keys`, `values` and `each` to repeatedly return the same order as each other. See "Algorithmic Complexity Attacks" in perlsec for details on why hash order is randomized. Aside from the guarantees provided here the exact details of Perl's hash algorithm and the hash traversal order are subject to change in any release of Perl. Tied hashes may behave differently to Perl's hashes with respect to changes in order on insertion and deletion of items.

As a side effect, calling `keys` resets the internal iterator of the HASH or ARRAY (see `each`). In particular, calling `keys` in void context resets the iterator with no other overhead.

Here is yet another way to print your environment:

```
my @keys = keys %ENV;
my @values = values %ENV;
while (@keys) {
    print pop(@keys), '=', pop(@values), "\n";
}
```

or how about sorted by key:

```
foreach my $key (sort(keys %ENV)) {
    print $key, '=', $ENV{$key}, "\n";
}
```

The returned values are copies of the original keys in the hash, so modifying them will not affect the original hash. Compare `values`.

To sort a hash by value, you'll need to use a `sort` function. Here's a descending numeric sort of a hash by its values:

```
foreach my $key (sort { $hash{$b} <=> $hash{$a} } keys %hash) {
    printf "%4d %s\n", $hash{$key}, $key;
}
```

Used as an lvalue, `keys` allows you to increase the number of hash buckets allocated for the given hash. This can gain you a measure of efficiency if you know the hash is going to get big. (This is similar to pre-extending an array by assigning a larger number to $#array.) If you say

```
keys %hash = 200;
```

then `%hash` will have at least 200 buckets allocated for it--256 of them, in fact, since it rounds up to the next power of two. These buckets will be retained even if you do `%hash = ()`, use `undef %hash` if you want to free the storage while `%hash` is still in scope. You can't shrink the number of buckets allocated for the hash using `keys` in this way (but you needn't worry about doing this by accident, as trying has no effect). `keys @array` in an lvalue context is a syntax error.

Starting with Perl 5.14, an experimental feature allowed `keys` to take a scalar expression. This experiment has been deemed unsuccessful, and was removed as of Perl 5.24.

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious syntax errors, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
use 5.012;  # so keys/values/each work on arrays
```

See also `each`, `values`, and `sort`.

kill SIGNAL, LIST
kill SIGNAL

Sends a signal to a list of processes. Returns the number of arguments that were successfully used to signal (which is not necessarily the same as the number of processes actually killed, e.g. where a process group is killed).

```
my $cnt = kill 'HUP', $child1, $child2;
kill 'KILL', @goners;
```

SIGNAL may be either a signal name (a string) or a signal number. A signal name may start with a `SIG` prefix, thus `FOO` and `SIGFOO` refer to the same signal. The string form of SIGNAL is recommended for portability because the same signal may have different numbers in different operating systems.

A list of signal names supported by the current platform can be found in `$Config{sig_name}`, which is provided by the `Config` module. See Config for more details.

A negative signal name is the same as a negative signal number, killing process groups instead of processes. For example, `kill '-KILL', $pgrp` and `kill -9, $pgrp` will send `SIGKILL` to the entire process group specified. That means you usually want to use positive not negative signals.

If SIGNAL is either the number 0 or the string `ZERO` (or `SIGZERO`), no signal is sent to the process, but `kill` checks whether it's *possible* to send a signal to it (that means, to be brief, that the process is owned by the same user, or we are the super-user). This is useful to check that a child process is still alive (even if only as a zombie) and hasn't changed its UID. See perlport for notes on the portability of this construct.

The behavior of kill when a *PROCESS* number is zero or negative depends on the operating system. For example, on POSIX-conforming systems, zero will signal the current process group, −1 will signal all processes, and any other negative PROCESS number will act as a negative signal number and kill the entire process group specified.

If both the SIGNAL and the PROCESS are negative, the results are undefined. A warning may be produced in a future version.

See "Signals" in perlipc for more details.

On some platforms such as Windows where the *fork*(2) system call is not available, Perl can be built to emulate `fork` at the interpreter level. This emulation has limitations related to kill that have to be considered, for code running on Windows and in code intended to be portable.

See perlfork for more details.

If there is no *LIST* of processes, no signal is sent, and the return value is 0. This form is sometimes used, however, because it causes tainting checks to be run. But see "Laundering and Detecting Tainted Data" in perlsec.

Portability issues: "kill" in perlport.

last LABEL
last EXPR
last The `last` command is like the `break` statement in C (as used in loops); it immediately exits the loop in question. If the LABEL is omitted, the command refers to the innermost enclosing loop. The `last EXPR` form, available starting in Perl 5.18.0, allows a label name to be computed at run time, and is otherwise identical to `last LABEL`. The `continue` block, if any, is not executed:

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

`last` cannot be used to exit a block that returns a value such as `eval {}`, `sub {}`, or `do {}`, and should not be used to exit a `grep` or `map` operation.

Note that a block by itself is semantically identical to a loop that executes once. Thus `last` can be used to effect an early exit out of such a block.

See also `continue` for an illustration of how `last`, `next`, and `redo` work.

Unlike most named operators, this has the same precedence as assignment. It is also exempt from the looks-like-a-function rule, so `last ("foo")."bar"` will cause "bar" to be part of the argument to `last`.

lc EXPR

lc Returns a lowercased version of EXPR. This is the internal function implementing the `\L` escape in double-quoted strings.

If EXPR is omitted, uses `$_`.

What gets returned depends on several factors:

If `use bytes` is in effect:

The results follow ASCII rules. Only the characters `A-Z` change, to `a-z` respectively.

Otherwise, if `use locale` for `LC_CTYPE` is in effect:

Respects current `LC_CTYPE` locale for code points < 256; and uses Unicode rules for the remaining code points (this last can only happen if the UTF8 flag is also set). See perllocale.

Starting in v5.20, Perl uses full Unicode rules if the locale is UTF-8. Otherwise, there is a deficiency in this scheme, which is that case changes that cross the 255/256 boundary are not well-defined. For example, the lower case of LATIN CAPITAL LETTER SHARP S (U+1E9E) in Unicode rules is U+00DF (on ASCII platforms). But under `use locale` (prior to v5.20 or not a UTF-8 locale), the lower case of U+1E9E is itself, because 0xDF may not be LATIN SMALL LETTER SHARP S in the current locale, and Perl has no way of knowing if that character even exists in the locale, much less what code point it is. Perl returns a result that is above 255 (almost always the input character unchanged), for all instances (and there aren't many) where the 255/256 boundary would otherwise be crossed; and starting in v5.22, it raises a locale warning.

Otherwise, If EXPR has the UTF8 flag set:

Unicode rules are used for the case change.

Otherwise, if `use feature 'unicode_strings'` or `use locale ':not_characters'` is in effect:

Unicode rules are used for the case change.

Otherwise:

ASCII rules are used for the case change. The lowercase of any character outside the ASCII range is the character itself.

lcfirst EXPR

lcfirst

Returns the value of EXPR with the first character lowercased. This is the internal function implementing the `\l` escape in double-quoted strings.

If EXPR is omitted, uses `$_`.

This function behaves the same way under various pragmas, such as in a locale, as `lc` does.

length EXPR

length

Returns the length in *characters* of the value of EXPR. If EXPR is omitted, returns the length of `$_`. If EXPR is undefined, returns `undef`.

This function cannot be used on an entire array or hash to find out how many elements these have. For that, use `scalar @array` and `scalar keys %hash`, respectively.

Like all Perl character operations, `length` normally deals in logical characters, not physical bytes. For how many bytes a string encoded as UTF-8 would take up, use `length(Encode::encode('UTF-8', EXPR))` (you'll have to `use Encode` first). See Encode and perlunicode.

__LINE__

A special token that compiles to the current line number.

link OLDFILE,NEWFILE

Creates a new filename linked to the old filename. Returns true for success, false otherwise.

Portability issues: "link" in perlport.

listen SOCKET,QUEUESIZE

Does the same thing that the *listen*(2) system call does. Returns true if it succeeded, false otherwise. See the example in "Sockets: Client/Server Communication" in perlipc.

local EXPR

You really probably want to be using `my` instead, because `local` isn't what most people think of as "local". See "Private Variables via *my()*" in perlsub for details.

A local modifies the listed variables to be local to the enclosing block, file, or eval. If more than one value is listed, the list must be placed in parentheses. See "Temporary Values via *local()*" in perlsub for details, including issues with tied arrays and hashes.

The `delete local EXPR` construct can also be used to localize the deletion of array/hash elements to the current block. See "Localized deletion of elements of composite types" in perlsub.

localtime EXPR

localtime

Converts a time as returned by the time function to a 9-element list with the time analyzed for the local time zone. Typically used as follows:

```
#     0    1    2     3     4    5     6     7     8
my ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
                                            localtime(time);
```

All list elements are numeric and come straight out of the C 'struct tm'. `$sec`, `$min`, and `$hour` are the seconds, minutes, and hours of the specified time.

`$mday` is the day of the month and `$mon` the month in the range `0..11`, with 0 indicating January and 11 indicating December. This makes it easy to get a month name from a list:

```
my @abbr = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
print "$abbr[$mon] $mday";
# $mon=9, $mday=18 gives "Oct 18"
```

`$year` contains the number of years since 1900. To get a 4-digit year write:

```
$year += 1900;
```

To get the last two digits of the year (e.g., "01" in 2001) do:

```
$year = sprintf("%02d", $year % 100);
```

`$wday` is the day of the week, with 0 indicating Sunday and 3 indicating Wednesday. `$yday` is the day of the year, in the range `0..364` (or `0..365` in leap years.)

`$isdst` is true if the specified time occurs during Daylight Saving Time, false otherwise.

If EXPR is omitted, `localtime` uses the current time (as returned by `time`).

In scalar context, `localtime` returns the *ctime* (3) value:

```
my $now_string = localtime;  # e.g., "Thu Oct 13 04:54:34 1994"
```

The format of this scalar value is **not** locale-dependent but built into Perl. For GMT instead of local time use the `gmtime` builtin. See also the `Time::Local` module (for converting seconds, minutes, hours, and such back to the integer value returned by `time`), and the POSIX module's `strftime` and `mktime` functions.

To get somewhat similar but locale-dependent date strings, set up your locale environment variables appropriately (please see perllocale) and try for example:

```
use POSIX qw(strftime);
my $now_string = strftime "%a %b %e %H:%M:%S %Y", localtime;
# or for GMT formatted appropriately for your locale:
my $now_string = strftime "%a %b %e %H:%M:%S %Y", gmtime;
```

Note that `%a` and `%b`, the short forms of the day of the week and the month of the year, may not necessarily be three characters wide.

The Time::gmtime and Time::localtime modules provide a convenient, by-name access mechanism to the `gmtime` and `localtime` functions, respectively.

For a comprehensive date and time representation look at the DateTime module on CPAN.

Portability issues: ‘‘localtime’’ in perlport.

lock THING
: This function places an advisory lock on a shared variable or referenced object contained in *THING* until the lock goes out of scope.

  The value returned is the scalar itself, if the argument is a scalar, or a reference, if the argument is a hash, array or subroutine.

  `lock` is a ‘‘weak keyword’’; this means that if you’ve defined a function by this name (before any calls to it), that function will be called instead. If you are not under `use threads::shared` this does nothing. See threads::shared.

log EXPR
log
: Returns the natural logarithm (base *e*) of EXPR. If EXPR is omitted, returns the log of `$_`. To get the log of another base, use basic algebra: The base-N log of a number is equal to the natural log of that number divided by the natural log of N. For example:

```
sub log10 {
    my $n = shift;
    return log($n)/log(10);
}
```

  See also `exp` for the inverse operation.

lstat FILEHANDLE
lstat EXPR
lstat DIRHANDLE
lstat
: Does the same thing as the `stat` function (including setting the special _ filehandle) but stats a symbolic link instead of the file the symbolic link points to. If symbolic links are unimplemented on your system, a normal `stat` is done. For much more detailed information, please see the documentation for `stat`.

  If EXPR is omitted, stats `$_`.

  Portability issues: ‘‘lstat’’ in perlport.

m//
: The match operator. See ‘‘Regexp Quote-Like Operators’’ in perlop.

map BLOCK LIST
map EXPR,LIST
: Evaluates the BLOCK or EXPR for each element of LIST (locally setting `$_` to each element) and returns the list value composed of the results of each such evaluation. In scalar context, returns the total number of elements so generated. Evaluates BLOCK or EXPR in list context, so each element of LIST may produce zero, one, or more elements in the returned value.

```
my @chars = map(chr, @numbers);
```

  translates a list of numbers to the corresponding characters.

```
my @squares = map { $_ * $_ } @numbers;
```

  translates a list of numbers to their squared values.

```
my @squares = map { $_ > 5 ? ($_ * $_) : () } @numbers;
```

  shows that number of returned elements can differ from the number of input elements. To omit an element, return an empty list (). This could also be achieved by writing

```
my @squares = map { $_ * $_ } grep { $_ > 5 } @numbers;
```

  which makes the intention more clear.

  Map always returns a list, which can be assigned to a hash such that the elements become key/value pairs. See perldata for more details.

```
my %hash = map { get_a_key_for($_) => $_ } @array;
```

  is just a funny way to write

```
my %hash;
foreach (@array) {
    $hash{get_a_key_for($_)} = $_;
}
```

Note that `$_` is an alias to the list value, so it can be used to modify the elements of the LIST. While this is useful and supported, it can cause bizarre results if the elements of LIST are not variables. Using a regular `foreach` loop for this purpose would be clearer in most cases. See also `grep` for a list composed of those items of the original list for which the BLOCK or EXPR evaluates to true.

`{` starts both hash references and blocks, so `map { ...` could be either the start of map BLOCK LIST or map EXPR, LIST. Because Perl doesn't look ahead for the closing `}` it has to take a guess at which it's dealing with based on what it finds just after the `{`. Usually it gets it right, but if it doesn't it won't realize something is wrong until it gets to the `}` and encounters the missing (or unexpected) comma. The syntax error will be reported close to the `}`, but you'll need to change something near the `{` such as using a unary + or semicolon to give Perl some help:

```
my %hash = map {  "\L$_" => 1  } @array # perl guesses EXPR. wrong
my %hash = map { +"\L$_" => 1  } @array # perl guesses BLOCK. right
my %hash = map {; "\L$_" => 1  } @array # this also works
my %hash = map { ("\L$_" => 1) } @array # as does this
my %hash = map {  lc($_) => 1  } @array # and this.
my %hash = map +( lc($_) => 1 ), @array # this is EXPR and works!

my %hash = map  ( lc($_), 1 ),   @array # evaluates to (1, @array)
```

or to force an anon hash constructor use `+{`:

```
my @hashes = map +{ lc($_) => 1 }, @array # EXPR, so needs
                                          # comma at end
```

to get a list of anonymous hashes each with only one entry apiece.

mkdir FILENAME,MASK
mkdir FILENAME
mkdir

Creates the directory specified by FILENAME, with permissions specified by MASK (as modified by `umask`). If it succeeds it returns true; otherwise it returns false and sets `$!` (errno). MASK defaults to 0777 if omitted, and FILENAME defaults to `$_` if omitted.

In general, it is better to create directories with a permissive MASK and let the user modify that with their `umask` than it is to supply a restrictive MASK and give the user no way to be more permissive. The exceptions to this rule are when the file or directory should be kept private (mail files, for instance). The documentation for `umask` discusses the choice of MASK in more detail.

Note that according to the POSIX 1003.1–1996 the FILENAME may have any number of trailing slashes. Some operating and filesystems do not get this right, so Perl automatically removes all trailing slashes to keep everyone happy.

To recursively create a directory structure, look at the `make_path` function of the File::Path module.

msgctl ID,CMD,ARG

Calls the System V IPC function *msgctl*(2). You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If CMD is `IPC_STAT`, then ARG must be a variable that will hold the returned `msqid_ds` structure. Returns like `ioctl`: the undefined value for error, `"0 but true"` for zero, or the actual return value otherwise. See also "SysV IPC" in perlipc and the documentation for `IPC::SysV` and `IPC::Semaphore`.

Portability issues: "msgctl" in perlport.

msgget KEY,FLAGS

Calls the System V IPC function *msgget*(2). Returns the message queue id, or `undef` on error. See also "SysV IPC" in perlipc and the documentation for `IPC::SysV` and `IPC::Msg`.

Portability issues: "msgget" in perlport.

msgrcv ID,VAR,SIZE,TYPE,FLAGS
: Calls the System V IPC function msgrcv to receive a message from message queue ID into variable VAR with a maximum message size of SIZE. Note that when a message is received, the message type as a native long integer will be the first thing in VAR, followed by the actual message. This packing may be opened with `unpack("l! a*")`. Taints the variable. Returns true if successful, false on error. See also "SysV IPC" in perlipc and the documentation for `IPC::SysV` and `IPC::Msg`.

  Portability issues: "msgrcv" in perlport.

msgsnd ID,MSG,FLAGS
: Calls the System V IPC function msgsnd to send the message MSG to the message queue ID. MSG must begin with the native long integer message type, be followed by the length of the actual message, and then finally the message itself. This kind of packing can be achieved with `pack("l! a*", $type, $message)`. Returns true if successful, false on error. See also "SysV IPC" in perlipc and the documentation for `IPC::SysV` and `IPC::Msg`.

  Portability issues: "msgsnd" in perlport.

my VARLIST
my TYPE VARLIST
my VARLIST : ATTRS
my TYPE VARLIST : ATTRS
: A `my` declares the listed variables to be local (lexically) to the enclosing block, file, or `eval`. If more than one variable is listed, the list must be placed in parentheses.

  The exact semantics and interface of TYPE and ATTRS are still evolving. TYPE may be a bareword, a constant declared with `use constant`, or `__PACKAGE__`. It is currently bound to the use of the fields pragma, and attributes are handled using the attributes pragma, or starting from Perl 5.8.0 also via the Attribute::Handlers module. See "Private Variables via *my()*" in perlsub for details.

  Note that with a parenthesised list, `undef` can be used as a dummy placeholder, for example to skip assignment of initial values:

```
my ( undef, $min, $hour ) = localtime;
```

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

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

  Note that if there were a `continue` block on the above, it would get executed even on discarded lines. If LABEL is omitted, the command refers to the innermost enclosing loop. The `next EXPR` form, available as of Perl 5.18.0, allows a label name to be computed at run time, being otherwise identical to `next LABEL`.

  `next` cannot be used to exit a block which returns a value such as `eval {}`, `sub {}`, or `do {}`, and should not be used to exit a `grep` or `map` operation.

  Note that a block by itself is semantically identical to a loop that executes once. Thus `next` will exit such a block early.

  See also `continue` for an illustration of how `last`, `next`, and `redo` work.

  Unlike most named operators, this has the same precedence as assignment. It is also exempt from the looks-like-a-function rule, so `next ("foo")."bar"` will cause "bar" to be part of the argument to `next`.

no MODULE VERSION LIST

no MODULE VERSION
no MODULE LIST
no MODULE
no VERSION
See the `use` function, of which `no` is the opposite.

oct EXPR
oct Interprets EXPR as an octal string and returns the corresponding value. (If EXPR happens to start off with `0x`, interprets it as a hex string. If EXPR starts off with `0b`, it is interpreted as a binary string. Leading whitespace is ignored in all three cases.) The following will handle decimal, binary, octal, and hex in standard Perl notation:

```
$val = oct($val) if $val =~ /^0/;
```

If EXPR is omitted, uses `$_`. To go the other way (produce a number in octal), use `sprintf` or `printf`:

```
my $dec_perms = (stat("filename"))[2] & 07777;
my $oct_perm_str = sprintf "%o", $perms;
```

The `oct` function is commonly used when a string such as `644` needs to be converted into a file mode, for example. Although Perl automatically converts strings into numbers as needed, this automatic conversion assumes base 10.

Leading white space is ignored without warning, as too are any trailing non-digits, such as a decimal point (`oct` only handles non-negative integers, not negative integers or floating point).

open FILEHANDLE,EXPR
open FILEHANDLE,MODE,EXPR
open FILEHANDLE,MODE,EXPR,LIST
open FILEHANDLE,MODE,REFERENCE
open FILEHANDLE
Opens the file whose filename is given by EXPR, and associates it with FILEHANDLE.

Simple examples to open a file for reading:

```
open(my $fh, "<", "input.txt")
    or die "Can't open < input.txt: $!";
```

and for writing:

```
open(my $fh, ">", "output.txt")
    or die "Can't open > output.txt: $!";
```

(The following is a comprehensive reference to `open`: for a gentler introduction you may consider perlopentut.)

If FILEHANDLE is an undefined scalar variable (or array or hash element), a new filehandle is autovivified, meaning that the variable is assigned a reference to a newly allocated anonymous filehandle. Otherwise if FILEHANDLE is an expression, its value is the real filehandle. (This is considered a symbolic reference, so `use strict "refs"` should *not* be in effect.)

If three (or more) arguments are specified, the open mode (including optional encoding) in the second argument are distinct from the filename in the third. If MODE is < or nothing, the file is opened for input. If MODE is >, the file is opened for output, with existing files first being truncated (''clobbered'') and nonexisting files newly created. If MODE is >>, the file is opened for appending, again being created if necessary.

You can put a + in front of the > or < to indicate that you want both read and write access to the file; thus +< is almost always preferred for read/write updates — the +> mode would clobber the file first. You can't usually use either read-write mode for updating textfiles, since they have variable-length records. See the **-i** switch in perlrun for a better approach. The file is created with permissions of `0666` modified by the process's `umask` value.

These various prefixes correspond to the *fopen*(3) modes of `r`, `r+`, `w`, `w+`, `a`, and `a+`.

In the one- and two-argument forms of the call, the mode and filename should be concatenated (in that

order), preferably separated by white space. You can — but shouldn't — omit the mode in these forms when that mode is <. It is safe to use the two-argument form of `open` if the filename argument is a known literal.

For three or more arguments if MODE is `|-`, the filename is interpreted as a command to which output is to be piped, and if MODE is `-|`, the filename is interpreted as a command that pipes output to us. In the two-argument (and one-argument) form, one should replace dash (-) with the command. See "Using *open()* for IPC" in perlipc for more examples of this. (You are not allowed to `open` to a command that pipes both in *and* out, but see IPC::Open2, IPC::Open3, and "Bidirectional Communication with Another Process" in perlipc for alternatives.)

In the form of pipe opens taking three or more arguments, if LIST is specified (extra arguments after the command name) then LIST becomes arguments to the command invoked if the platform supports it. The meaning of `open` with more than three arguments for non-pipe modes is not yet defined, but experimental "layers" may give extra LIST arguments meaning.

In the two-argument (and one-argument) form, opening `<-` or `-` opens STDIN and opening `>-` opens STDOUT.

You may (and usually should) use the three-argument form of open to specify I/O layers (sometimes referred to as "disciplines") to apply to the handle that affect how the input and output are processed (see open and PerlIO for more details). For example:

```
open(my $fh, "<:encoding(UTF-8)", $filename)
  || die "Can't open UTF-8 encoded $filename: $!";
```

opens the UTF8-encoded file containing Unicode characters; see perluniintro. Note that if layers are specified in the three-argument form, then default layers stored in ${^OPEN} (see perlvar; usually set by the open pragma or the switch `-CioD`) are ignored. Those layers will also be ignored if you specify a colon with no name following it. In that case the default layer for the operating system (:raw on Unix, :crlf on Windows) is used.

Open returns nonzero on success, the undefined value otherwise. If the `open` involved a pipe, the return value happens to be the pid of the subprocess.

On some systems (in general, DOS- and Windows-based systems) `binmode` is necessary when you're not working with a text file. For the sake of portability it is a good idea always to use it when appropriate, and never to use it when it isn't appropriate. Also, people can set their I/O to be by default UTF8-encoded Unicode, not bytes.

When opening a file, it's seldom a good idea to continue if the request failed, so `open` is frequently used with `die`. Even if `die` won't do what you want (say, in a CGI script, where you want to format a suitable error message (but there are modules that can help with that problem)) always check the return value from opening a file.

The filehandle will be closed when its reference count reaches zero. If it is a lexically scoped variable declared with `my`, that usually means the end of the enclosing scope. However, this automatic close does not check for errors, so it is better to explicitly close filehandles, especially those used for writing:

```
close($handle)
   || warn "close failed: $!";
```

An older style is to use a bareword as the filehandle, as

```
open(FH, "<", "input.txt")
   or die "Can't open < input.txt: $!";
```

Then you can use `FH` as the filehandle, in `close FH` and `<FH>` and so on. Note that it's a global variable, so this form is not recommended in new code.

As a shortcut a one-argument call takes the filename from the global scalar variable of the same name as the filehandle:

```
$ARTICLE = 100;
open(ARTICLE) or die "Can't find article $ARTICLE: $!\n";
```

Here `$ARTICLE` must be a global (package) scalar variable – not one declared with `my` or `state`.

As a special case the three-argument form with a read/write mode and the third argument being `undef`:

```
open(my $tmp, "+>", undef) or die ...
```

opens a filehandle to a newly created empty anonymous temporary file. (This happens under any mode, which makes `+>` the only useful and sensible mode to use.) You will need to `seek` to do the reading.

Perl is built using PerlIO by default. Unless you've changed this (such as building Perl with `Configure -Uuseperlio`), you can open filehandles directly to Perl scalars via:

```
open(my $fh, ">", \$variable) || ..
```

To (re)open `STDOUT` or `STDERR` as an in-memory file, close it first:

```
close STDOUT;
open(STDOUT, ">", \$variable)
    or die "Can't open STDOUT: $!";
```

See perliol for detailed info on PerlIO.

General examples:

```
open(my $log, ">>", "/usr/spool/news/twitlog");
# if the open fails, output is discarded

open(my $dbase, "+<", "dbase.mine")      # open for update
    or die "Can't open 'dbase.mine' for update: $!";

open(my $dbase, "+<dbase.mine")          # ditto
    or die "Can't open 'dbase.mine' for update: $!";

open(my $article_fh, "-|", "caesar <$article")  # decrypt
                                                # article
    or die "Can't start caesar: $!";

open(my $article_fh, "caesar <$article |")      # ditto
    or die "Can't start caesar: $!";

open(my $out_fh, "|-", "sort >Tmp$$")    # $$ is our process id
    or die "Can't start sort: $!";

# in-memory files
open(my $memory, ">", \$var)
    or die "Can't open memory file: $!";
print $memory "foo!\n";               # output will appear in $var
```

You may also, in the Bourne shell tradition, specify an EXPR beginning with `>&`, in which case the rest of the string is interpreted as the name of a filehandle (or file descriptor, if numeric) to be duped (as in *dup*(2)) and opened. You may use `&` after `>`, `>>`, `<`, `+>`, `+>>`, and `+<`. The mode you specify should match the mode of the original filehandle. (Duping a filehandle does not take into account any existing contents of IO buffers.) If you use the three-argument form, then you can pass either a number, the name of a filehandle, or the normal "reference to a glob".

Here is a script that saves, redirects, and restores `STDOUT` and `STDERR` using various methods:

```
#!/usr/bin/perl
open(my $oldout, ">&STDOUT")     or die "Can't dup STDOUT: $!";
open(OLDERR,     ">&", \*STDERR) or die "Can't dup STDERR: $!";

open(STDOUT, '>', "foo.out") or die "Can't redirect STDOUT: $!";
open(STDERR, ">&STDOUT")     or die "Can't dup STDOUT: $!";
```

```
select STDERR; $| = 1;  # make unbuffered
select STDOUT; $| = 1;  # make unbuffered

print STDOUT "stdout 1\n";  # this works for
print STDERR "stderr 1\n";  # subprocesses too

open(STDOUT, ">&", $oldout) or die "Can't dup \$oldout: $!";
open(STDERR, ">&OLDERR")    or die "Can't dup OLDERR: $!";

print STDOUT "stdout 2\n";
print STDERR "stderr 2\n";
```

If you specify `'<&=X'`, where `X` is a file descriptor number or a filehandle, then Perl will do an equivalent of C's *fdopen* (3) of that file descriptor (and not call *dup* (2)); this is more parsimonious of file descriptors. For example:

```
# open for input, reusing the fileno of $fd
open(my $fh, "<&=", $fd)
```

or

```
open(my $fh, "<&=$fd")
```

or

```
# open for append, using the fileno of $oldfh
open(my $fh, ">>&=", $oldfh)
```

Being parsimonious on filehandles is also useful (besides being parsimonious) for example when something is dependent on file descriptors, like for example locking using `flock`. If you do just `open(my $A, ">>&", $B)`, the filehandle `$A` will not have the same file descriptor as `$B`, and therefore `flock($A)` will not `flock($B)` nor vice versa. But with `open(my $A, ">>&=", $B)`, the filehandles will share the same underlying system file descriptor.

Note that under Perls older than 5.8.0, Perl uses the standard C library's' *fdopen* (3) to implement the = functionality. On many Unix systems, *fdopen* (3) fails when file descriptors exceed a certain value, typically 255. For Perls 5.8.0 and later, PerlIO is (most often) the default.

You can see whether your Perl was built with PerlIO by running `perl -V:useperlio`. If it says `'define'`, you have PerlIO; otherwise you don't.

If you open a pipe on the command − (that is, specify either `|-` or `-|` with the one– or two-argument forms of `open`), an implicit `fork` is done, so `open` returns twice: in the parent process it returns the pid of the child process, and in the child process it returns (a defined) `0`. Use `defined($pid)` or `//` to determine whether the open was successful.

For example, use either

```
my $child_pid = open(my $from_kid, "-|") // die "Can't fork: $!";
```

or

```
my $child_pid = open(my $to_kid,   "|-") // die "Can't fork: $!";
```

followed by

```
if ($child_pid) {
    # am the parent:
    # either write $to_kid or else read $from_kid
    ...
   waitpid $child_pid, 0;
} else {
    # am the child; use STDIN/STDOUT normally
    ...
    exit;
}
```

The filehandle behaves normally for the parent, but I/O to that filehandle is piped from/to the STDOUT/STDIN of the child process. In the child process, the filehandle isn't opened — I/O happens from/to the new STDOUT/STDIN. Typically this is used like the normal piped open when you want to exercise more control over just how the pipe command gets executed, such as when running setuid and you don't want to have to scan shell commands for metacharacters.

The following blocks are more or less equivalent:

```
open(my $fh, "|tr '[a-z]' '[A-Z]'");
open(my $fh, "|-", "tr '[a-z]' '[A-Z]'");
open(my $fh, "|-") || exec 'tr', '[a-z]', '[A-Z]';
open(my $fh, "|-", "tr", '[a-z]', '[A-Z]');

open(my $fh, "cat -n '$file'|");
open(my $fh, "-|", "cat -n '$file'");
open(my $fh, "-|") || exec "cat", "-n", $file;
open(my $fh, "-|", "cat", "-n", $file);
```

The last two examples in each block show the pipe as "list form", which is not yet supported on all platforms. A good rule of thumb is that if your platform has a real `fork` (in other words, if your platform is Unix, including Linux and MacOS X), you can use the list form. You would want to use the list form of the pipe so you can pass literal arguments to the command without risk of the shell interpreting any shell metacharacters in them. However, this also bars you from opening pipes to commands that intentionally contain shell metacharacters, such as:

```
open(my $fh, "|cat -n | expand -4 | lpr")
    || die "Can't open pipeline to lpr: $!";
```

See "Safe Pipe Opens" in perlipc for more examples of this.

Perl will attempt to flush all files opened for output before any operation that may do a fork, but this may not be supported on some platforms (see perlport). To be safe, you may need to set `$|` (`$AUTOFLUSH` in English) or call the `autoflush` method of `IO::Handle` on any open handles.

On systems that support a close-on-exec flag on files, the flag will be set for the newly opened file descriptor as determined by the value of `$^F`. See "$^F" in perlvar.

Closing any piped filehandle causes the parent process to wait for the child to finish, then returns the status value in `$?` and `${^CHILD_ERROR_NATIVE}`.

The filename passed to the one- and two-argument forms of `open` will have leading and trailing whitespace deleted and normal redirection characters honored. This property, known as "magic open", can often be used to good effect. A user could specify a filename of *"rsh cat file |"*, or you could change certain filenames as needed:

```
$filename =~ s/(.*\.gz)\s*$/gzip -dc < $1|/;
open(my $fh, $filename) or die "Can't open $filename: $!";
```

Use the three-argument form to open a file with arbitrary weird characters in it,

```
open(my $fh, "<", $file)
    || die "Can't open $file: $!";
```

otherwise it's necessary to protect any leading and trailing whitespace:

```
$file =~ s#^(\s)#./$1#;
open(my $fh, "< $file\0")
    || die "Can't open $file: $!";
```

(this may not work on some bizarre filesystems). One should conscientiously choose between the *magic* and *three-argument* form of `open`:

```
open(my $in, $ARGV[0]) || die "Can't open $ARGV[0]: $!";
```

will allow the user to specify an argument of the form `"rsh cat file |"`, but will not work on a filename that happens to have a trailing space, while

```
open(my $in, "<", $ARGV[0])
    || die "Can't open $ARGV[0]: $!";
```

will have exactly the opposite restrictions. (However, some shells support the syntax `perl your_program.pl <( rsh cat file )`, which produces a filename that can be opened normally.)

If you want a "real" C *open*(2), then you should use the `sysopen` function, which involves no such magic (but uses different filemodes than Perl `open`, which corresponds to C *fopen*(3)). This is another way to protect your filenames from interpretation. For example:

```
use IO::Handle;
sysopen(my $fh, $path, O_RDWR|O_CREAT|O_EXCL)
    or die "Can't open $path: $!";
$fh->autoflush(1);
print $fh "stuff $$\n";
seek($fh, 0, 0);
print "File contains: ", readline($fh);
```

See `seek` for some details about mixing reading and writing.

Portability issues: "open" in perlport.

opendir DIRHANDLE,EXPR

Opens a directory named EXPR for processing by `readdir`, `telldir`, `seekdir`, `rewinddir`, and `closedir`. Returns true if successful. DIRHANDLE may be an expression whose value can be used as an indirect dirhandle, usually the real dirhandle name. If DIRHANDLE is an undefined scalar variable (or array or hash element), the variable is assigned a reference to a new anonymous dirhandle; that is, it's autovivified. DIRHANDLEs have their own namespace separate from FILEHANDLEs.

See the example at `readdir`.

ord EXPR

ord Returns the numeric value of the first character of EXPR. If EXPR is an empty string, returns 0. If EXPR is omitted, uses `$_`. (Note *character*, not byte.)

For the reverse, see `chr`. See perlunicode for more about Unicode.

our VARLIST

our TYPE VARLIST

our VARLIST : ATTRS

our TYPE VARLIST : ATTRS

`our` makes a lexical alias to a package (i.e. global) variable of the same name in the current package for use within the current lexical scope.

`our` has the same scoping rules as `my` or `state`, meaning that it is only valid within a lexical scope. Unlike `my` and `state`, which both declare new (lexical) variables, `our` only creates an alias to an existing variable: a package variable of the same name.

This means that when `use strict 'vars'` is in effect, `our` lets you use a package variable without qualifying it with the package name, but only within the lexical scope of the `our` declaration. This applies immediately — even within the same statement.

```
package Foo;
use strict;

$Foo::foo = 23;

{
    our $foo;   # alias to $Foo::foo
    print $foo; # prints 23
}

print $Foo::foo; # prints 23

print $foo; # ERROR: requires explicit package name
```

This works even if the package variable has not been used before, as package variables spring into existence when first used.

```
package Foo;
use strict;

our $foo = 23;   # just like $Foo::foo = 23

print $Foo::foo; # prints 23
```

Because the variable becomes legal immediately under `use strict 'vars'`, so long as there is no variable with that name is already in scope, you can then reference the package variable again even within the same statement.

```
package Foo;
use strict;

my  $foo = $foo; # error, undeclared $foo on right-hand side
our $foo = $foo; # no errors
```

If more than one variable is listed, the list must be placed in parentheses.

```
our($bar, $baz);
```

An `our` declaration declares an alias for a package variable that will be visible across its entire lexical scope, even across package boundaries. The package in which the variable is entered is determined at the point of the declaration, not at the point of use. This means the following behavior holds:

```
package Foo;
our $bar;      # declares $Foo::bar for rest of lexical scope
$bar = 20;

package Bar;
print $bar;    # prints 20, as it refers to $Foo::bar
```

Multiple `our` declarations with the same name in the same lexical scope are allowed if they are in different packages. If they happen to be in the same package, Perl will emit warnings if you have asked for them, just like multiple `my` declarations. Unlike a second `my` declaration, which will bind the name to a fresh variable, a second `our` declaration in the same package, in the same scope, is merely redundant.

```
use warnings;
package Foo;
our $bar;      # declares $Foo::bar for rest of lexical scope
$bar = 20;

package Bar;
our $bar = 30; # declares $Bar::bar for rest of lexical scope
print $bar;    # prints 30
```

```
our $bar;      # emits warning but has no other effect
print $bar;    # still prints 30
```

An `our` declaration may also have a list of attributes associated with it.

The exact semantics and interface of TYPE and ATTRS are still evolving. TYPE is currently bound to the use of the fields pragma, and attributes are handled using the attributes pragma, or, starting from Perl 5.8.0, also via the Attribute::Handlers module. See "Private Variables via *my()*" in perlsub for details.

Note that with a parenthesised list, `undef` can be used as a dummy placeholder, for example to skip assignment of initial values:

```
our ( undef, $min, $hour ) = localtime;
```

`our` differs from `use vars`, which allows use of an unqualified name *only* within the affected package, but across scopes.

pack TEMPLATE,LIST

Takes a LIST of values and converts it into a string using the rules given by the TEMPLATE. The resulting string is the concatenation of the converted values. Typically, each converted value looks like its machine-level representation. For example, on 32-bit machines an integer may be represented by a sequence of 4 bytes, which will in Perl be presented as a string that's 4 characters long.

See perlpacktut for an introduction to this function.

The TEMPLATE is a sequence of characters that give the order and type of values, as follows:

```
a  A string with arbitrary binary data, will be null padded.
A  A text (ASCII) string, will be space padded.
Z  A null-terminated (ASCIZ) string, will be null padded.

b  A bit string (ascending bit order inside each byte,
   like vec()).
B  A bit string (descending bit order inside each byte).
h  A hex string (low nybble first).
H  A hex string (high nybble first).

c  A signed char (8-bit) value.
C  An unsigned char (octet) value.
W  An unsigned char value (can be greater than 255).

s  A signed short (16-bit) value.
S  An unsigned short value.

l  A signed long (32-bit) value.
L  An unsigned long value.

q  A signed quad (64-bit) value.
Q  An unsigned quad value.
     (Quads are available only if your system supports 64-bit
      integer values _and_ if Perl has been compiled to support
      those.  Raises an exception otherwise.)

i  A signed integer value.
I  A unsigned integer value.
     (This 'integer' is _at_least_ 32 bits wide.  Its exact
      size depends on what a local C compiler calls 'int'.)

n  An unsigned short (16-bit) in "network" (big-endian) order.
N  An unsigned long (32-bit) in "network" (big-endian) order.
v  An unsigned short (16-bit) in "VAX" (little-endian) order.
V  An unsigned long (32-bit) in "VAX" (little-endian) order.
```

```
j  A Perl internal signed integer value (IV).
J  A Perl internal unsigned integer value (UV).

f  A single-precision float in native format.
d  A double-precision float in native format.

F  A Perl internal floating-point value (NV) in native format
D  A float of long-double precision in native format.
     (Long doubles are available only if your system supports
      long double values _and_ if Perl has been compiled to
      support those.  Raises an exception otherwise.
      Note that there are different long double formats.)

p  A pointer to a null-terminated string.
P  A pointer to a structure (fixed-length string).

u  A uuencoded string.
U  A Unicode character number.  Encodes to a character in char-
   acter mode and UTF-8 (or UTF-EBCDIC in EBCDIC platforms) in
   byte mode.

w  A BER compressed integer (not an ASN.1 BER, see perlpacktut
   for details).  Its bytes represent an unsigned integer in
   base 128, most significant digit first, with as few digits
   as possible.  Bit eight (the high bit) is set on each byte
   except the last.

x  A null byte (a.k.a ASCII NUL, "\000", chr(0))
X  Back up a byte.
@  Null-fill or truncate to absolute position, counted from the
   start of the innermost ()-group.
.  Null-fill or truncate to absolute position specified by
   the value.
(  Start of a ()-group.
```

One or more modifiers below may optionally follow certain letters in the TEMPLATE (the second column lists letters for which the modifier is valid):

```
!   sSlLiI     Forces native (short, long, int) sizes instead
               of fixed (16-/32-bit) sizes.

!   xX         Make x and X act as alignment commands.

!   nNvV       Treat integers as signed instead of unsigned.

!   @.         Specify position as byte offset in the internal
               representation of the packed string.  Efficient
               but dangerous.

>   sSiIlLqQ   Force big-endian byte-order on the type.
    jJfFdDpP   (The "big end" touches the construct.)

<   sSiIlLqQ   Force little-endian byte-order on the type.
    jJfFdDpP   (The "little end" touches the construct.)
```

The > and < modifiers can also be used on () groups to force a particular byte-order on all components in that group, including all its subgroups.

The following rules apply:

- Each letter may optionally be followed by a number indicating the repeat count. A numeric repeat count may optionally be enclosed in brackets, as in `pack("C[80]", @arr)`. The repeat count gobbles that many values from the LIST when used with all format types other than `a`, `A`, `Z`, `b`, `B`, `h`, `H`, `@`, `.`, `x`, `X`, and `P`, where it means something else, described below. Supplying a `*` for the repeat count instead of a number means to use however many items are left, except for:

  - `@`, `x`, and `X`, where it is equivalent to `0`.
  - `<.>`, where it means relative to the start of the string.
  - `u`, where it is equivalent to 1 (or 45, which here is equivalent).

  One can replace a numeric repeat count with a template letter enclosed in brackets to use the packed byte length of the bracketed template for the repeat count.

  For example, the template `x[L]` skips as many bytes as in a packed long, and the template `"$t X[$t] $t"` unpacks twice whatever `$t` (when variable-expanded) unpacks. If the template in brackets contains alignment commands (such as `x![d]`), its packed length is calculated as if the start of the template had the maximal possible alignment.

  When used with `Z`, a `*` as the repeat count is guaranteed to add a trailing null byte, so the resulting string is always one byte longer than the byte length of the item itself.

  When used with `@`, the repeat count represents an offset from the start of the innermost `()` group.

  When used with `.`, the repeat count determines the starting position to calculate the value offset as follows:

  - If the repeat count is `0`, it's relative to the current position.
  - If the repeat count is `*`, the offset is relative to the start of the packed string.
  - And if it's an integer *n*, the offset is relative to the start of the *n*th innermost `( )` group, or to the start of the string if *n* is bigger then the group level.

  The repeat count for `u` is interpreted as the maximal number of bytes to encode per line of output, with 0, 1 and 2 replaced by 45. The repeat count should not be more than 65.

- The `a`, `A`, and `Z` types gobble just one value, but pack it as a string of length count, padding with nulls or spaces as needed. When unpacking, `A` strips trailing whitespace and nulls, `Z` strips everything after the first null, and `a` returns data with no stripping at all.

  If the value to pack is too long, the result is truncated. If it's too long and an explicit count is provided, `Z` packs only `$count-1` bytes, followed by a null byte. Thus `Z` always packs a trailing null, except when the count is 0.

- Likewise, the `b` and `B` formats pack a string that's that many bits long. Each such format generates 1 bit of the result. These are typically followed by a repeat count like `B8` or `B64`.

  Each result bit is based on the least-significant bit of the corresponding input character, i.e., on `ord($char)%2`. In particular, characters `"0"` and `"1"` generate bits 0 and 1, as do characters `"\000"` and `"\001"`.

  Starting from the beginning of the input string, each 8−tuple of characters is converted to 1 character of output. With format `b`, the first character of the 8−tuple determines the least-significant bit of a character; with format `B`, it determines the most-significant bit of a character.

  If the length of the input string is not evenly divisible by 8, the remainder is packed as if the input string were padded by null characters at the end. Similarly during unpacking, "extra" bits are ignored.

  If the input string is longer than needed, remaining characters are ignored.

  A `*` for the repeat count uses all characters of the input field. On unpacking, bits are converted to a string of `0`s and `1`s.

- The `h` and `H` formats pack a string that many nybbles (4−bit groups, representable as hexadecimal digits, `"0".."9" "a".."f"`) long.

  For each such format, `pack` generates 4 bits of result. With non-alphabetical characters, the

result is based on the 4 least-significant bits of the input character, i.e., on `ord($char)%16`. In particular, characters `"0"` and `"1"` generate nybbles 0 and 1, as do bytes `"\000"` and `"\001"`. For characters `"a".."f"` and `"A".."F"`, the result is compatible with the usual hexadecimal digits, so that `"a"` and `"A"` both generate the nybble `0xA==10`. Use only these specific hex characters with this format.

Starting from the beginning of the template to `pack`, each pair of characters is converted to 1 character of output. With format `h`, the first character of the pair determines the least-significant nybble of the output character; with format `H`, it determines the most-significant nybble.

If the length of the input string is not even, it behaves as if padded by a null character at the end. Similarly, ''extra'' nybbles are ignored during unpacking.

If the input string is longer than needed, extra characters are ignored.

A `*` for the repeat count uses all characters of the input field. For `unpack`, nybbles are converted to a string of hexadecimal digits.

- The `p` format packs a pointer to a null-terminated string. You are responsible for ensuring that the string is not a temporary value, as that could potentially get deallocated before you got around to using the packed result. The `P` format packs a pointer to a structure of the size indicated by the length. A null pointer is created if the corresponding value for `p` or `P` is `undef`; similarly with `unpack`, where a null pointer unpacks into `undef`.

  If your system has a strange pointer size — meaning a pointer is neither as big as an int nor as big as a long — it may not be possible to pack or unpack pointers in big− or little-endian byte order. Attempting to do so raises an exception.

- The `/` template character allows packing and unpacking of a sequence of items where the packed structure contains a packed item count followed by the packed items themselves. This is useful when the structure you're unpacking has encoded the sizes or repeat counts for some of its fields within the structure itself as separate fields.

  For `pack`, you write *length-item*`/`*sequence-item*, and the *length-item* describes how the length value is packed. Formats likely to be of most use are integer-packing ones like `n` for Java strings, `w` for ASN.1 or SNMP, and `N` for Sun XDR.

  For `pack`, *sequence-item* may have a repeat count, in which case the minimum of that and the number of available items is used as the argument for *length-item*. If it has no repeat count or uses a '*', the number of available items is used.

  For `unpack`, an internal stack of integer arguments unpacked so far is used. You write `/`*sequence-item* and the repeat count is obtained by popping off the last element from the stack. The *sequence-item* must not have a repeat count.

  If *sequence-item* refers to a string type (`"A"`, `"a"`, or `"Z"`), the *length-item* is the string length, not the number of strings. With an explicit repeat count for pack, the packed string is adjusted to that length. For example:

```
This code:                              gives this result:

unpack("W/a", "\004Gurusamy")           ("Guru")
unpack("a3/A A*", "007 Bond  J ")       (" Bond", "J")
unpack("a3 x2 /A A*", "007: Bond, J.") ("Bond, J", ".")

pack("n/a* w/a","hello,","world")      "\000\006hello,\005world"
pack("a/W2", ord("a") .. ord("z"))     "2ab"
```

  The *length-item* is not returned explicitly from `unpack`.

  Supplying a count to the *length-item* format letter is only useful with `A`, `a`, or `Z`. Packing with a *length-item* of `a` or `Z` may introduce `"\000"` characters, which Perl does not regard as legal in numeric strings.

- The integer types `s`, `S`, `l`, and `L` may be followed by a `!` modifier to specify native shorts or longs. As shown in the example above, a bare `l` means exactly 32 bits, although the native `long`

as seen by the local C compiler may be larger. This is mainly an issue on 64−bit platforms. You can see whether using `!` makes any difference this way:

```
printf "format s is %d, s! is %d\n",
    length pack("s"), length pack("s!");

printf "format l is %d, l! is %d\n",
    length pack("l"), length pack("l!");
```

`i!` and `I!` are also allowed, but only for completeness' sake: they are identical to `i` and `I`.

The actual sizes (in bytes) of native shorts, ints, longs, and long longs on the platform where Perl was built are also available from the command line:

```
$ perl -V:{short,int,long{,long}}size
shortsize='2';
intsize='4';
longsize='4';
longlongsize='8';
```

or programmatically via the `Config` module:

```
use Config;
print $Config{shortsize},    "\n";
print $Config{intsize},      "\n";
print $Config{longsize},     "\n";
print $Config{longlongsize}, "\n";
```

`$Config{longlongsize}` is undefined on systems without long long support.

- The integer formats `s`, `S`, `i`, `I`, `l`, `L`, `j`, and `J` are inherently non-portable between processors and operating systems because they obey native byteorder and endianness. For example, a 4−byte integer 0x12345678 (305419896 decimal) would be ordered natively (arranged in and handled by the CPU registers) into bytes as

  ```
  0x12 0x34 0x56 0x78  # big-endian
  0x78 0x56 0x34 0x12  # little-endian
  ```

  Basically, Intel and VAX CPUs are little-endian, while everybody else, including Motorola m68k/88k, PPC, Sparc, HP PA, Power, and Cray, are big-endian. Alpha and MIPS can be either: Digital/Compaq uses (well, used) them in little-endian mode, but SGI/Cray uses them in big-endian mode.

  The names *big-endian* and *little-endian* are comic references to the egg-eating habits of the little-endian Lilliputians and the big-endian Blefuscudians from the classic Jonathan Swift satire, *Gulliver's Travels*. This entered computer lingo via the paper ''On Holy Wars and a Plea for Peace'' by Danny Cohen, USC/ISI IEN 137, April 1, 1980.

  Some systems may have even weirder byte orders such as

  ```
  0x56 0x78 0x12 0x34
  0x34 0x12 0x78 0x56
  ```

  These are called mid-endian, middle-endian, mixed-endian, or just weird.

  You can determine your system endianness with this incantation:

  ```
  printf("%#02x ", $_) for unpack("W*", pack L=>0x12345678);
  ```

  The byteorder on the platform where Perl was built is also available via Config:

  ```
  use Config;
  print "$Config{byteorder}\n";
  ```

  or from the command line:

  ```
  $ perl -V:byteorder
  ```

  Byteorders `"1234"` and `"12345678"` are little-endian; `"4321"` and `"87654321"` are big-

endian. Systems with multiarchitecture binaries will have `"ffff"`, signifying that static information doesn't work, one must use runtime probing.

For portably packed integers, either use the formats `n`, `N`, `v`, and `V` or else use the `>` and `<` modifiers described immediately below. See also perlport.

- Also floating point numbers have endianness. Usually (but not always) this agrees with the integer endianness. Even though most platforms these days use the IEEE 754 binary format, there are differences, especially if the long doubles are involved. You can see the `Config` variables `doublekind` and `longdblkind` (also `doublesize`, `longdblsize`): the "kind" values are enums, unlike `byteorder`.

  Portability-wise the best option is probably to keep to the IEEE 754 64–bit doubles, and of agreed-upon endianness. Another possibility is the `"%a"`) format of `printf`.

- Starting with Perl 5.10.0, integer and floating-point formats, along with the `p` and `P` formats and `()` groups, may all be followed by the `>` or `<` endianness modifiers to respectively enforce big– or little-endian byte-order. These modifiers are especially useful given how `n`, `N`, `v`, and `V` don't cover signed integers, 64–bit integers, or floating-point values.

  Here are some concerns to keep in mind when using an endianness modifier:

  - Exchanging signed integers between different platforms works only when all platforms store them in the same format. Most platforms store signed integers in two's-complement notation, so usually this is not an issue.
  - The `>` or `<` modifiers can only be used on floating-point formats on big– or little-endian machines. Otherwise, attempting to use them raises an exception.
  - Forcing big– or little-endian byte-order on floating-point values for data exchange can work only if all platforms use the same binary representation such as IEEE floating-point. Even if all platforms are using IEEE, there may still be subtle differences. Being able to use `>` or `<` on floating-point values can be useful, but also dangerous if you don't know exactly what you're doing. It is not a general way to portably store floating-point values.
  - When using `>` or `<` on a `()` group, this affects all types inside the group that accept byte-order modifiers, including all subgroups. It is silently ignored for all other types. You are not allowed to override the byte-order within a group that already has a byte-order modifier suffix.

- Real numbers (floats and doubles) are in native machine format only. Due to the multiplicity of floating-point formats and the lack of a standard "network" representation for them, no facility for interchange has been made. This means that packed floating-point data written on one machine may not be readable on another, even if both use IEEE floating-point arithmetic (because the endianness of the memory representation is not part of the IEEE spec). See also perlport.

  If you know *exactly* what you're doing, you can use the `>` or `<` modifiers to force big– or little-endian byte-order on floating-point values.

  Because Perl uses doubles (or long doubles, if configured) internally for all numeric calculation, converting from double into float and thence to double again loses precision, so `unpack("f", pack("f", $foo))` will not in general equal `$foo`.

- Pack and unpack can operate in two modes: character mode (`C0` mode) where the packed string is processed per character, and UTF–8 byte mode (`U0` mode) where the packed string is processed in its UTF–8–encoded Unicode form on a byte-by-byte basis. Character mode is the default unless the format string starts with `U`. You can always switch mode mid-format with an explicit `C0` or `U0` in the format. This mode remains in effect until the next mode change, or until the end of the `()` group it (directly) applies to.

  Using `C0` to get Unicode characters while using `U0` to get *non*–Unicode bytes is not necessarily obvious. Probably only the first of these is what you want:

```
$ perl -CS -E 'say "\x{3B1}\x{3C9}"' |
  perl -CS -ne 'printf "%v04X\n", $_ for unpack("C0A*", $_)'
03B1.03C9
$ perl -CS -E 'say "\x{3B1}\x{3C9}"' |
  perl -CS -ne 'printf "%v02X\n", $_ for unpack("U0A*", $_)'
CE.B1.CF.89
$ perl -CS -E 'say "\x{3B1}\x{3C9}"' |
  perl -C0 -ne 'printf "%v02X\n", $_ for unpack("C0A*", $_)'
CE.B1.CF.89
$ perl -CS -E 'say "\x{3B1}\x{3C9}"' |
  perl -C0 -ne 'printf "%v02X\n", $_ for unpack("U0A*", $_)'
C3.8E.C2.B1.C3.8F.C2.89
```

Those examples also illustrate that you should not try to use `pack`/`unpack` as a substitute for the Encode module.

- You must yourself do any alignment or padding by inserting, for example, enough `"x"`es while packing. There is no way for `pack` and `unpack` to know where characters are going to or coming from, so they handle their output and input as flat sequences of characters.
- A `()` group is a sub-TEMPLATE enclosed in parentheses. A group may take a repeat count either as postfix, or for `unpack`, also via the `/` template character. Within each repetition of a group, positioning with `@` starts over at 0. Therefore, the result of

  ```
  pack("@1A((@2A)@3A)", qw[X Y Z])
  ```

  is the string `"\0X\0\0YZ"`.
- `x` and `X` accept the `!` modifier to act as alignment commands: they jump forward or back to the closest position aligned at a multiple of `count` characters. For example, to `pack` or `unpack` a C structure like

  ```
  struct {
      char   c;    /* one signed, 8-bit character */
      double d;
      char   cc[2];
  }
  ```

  one may need to use the template `c x![d] d c[2]`. This assumes that doubles must be aligned to the size of double.

  For alignment commands, a `count` of 0 is equivalent to a `count` of 1; both are no-ops.
- `n`, `N`, `v` and `V` accept the `!` modifier to represent signed 16–/32–bit integers in big–/little–endian order. This is portable only when all platforms sharing packed data use the same binary representation for signed integers; for example, when all platforms use two's-complement representation.
- Comments can be embedded in a TEMPLATE using `#` through the end of line. White space can separate pack codes from each other, but modifiers and repeat counts must follow immediately. Breaking complex templates into individual line-by-line components, suitably annotated, can do as much to improve legibility and maintainability of pack/unpack formats as `/x` can for complicated pattern matches.
- If TEMPLATE requires more arguments than `pack` is given, `pack` assumes additional `""` arguments. If TEMPLATE requires fewer arguments than given, extra arguments are ignored.
- Attempting to pack the special floating point values `Inf` and `NaN` (infinity, also in negative, and not-a-number) into packed integer values (like `"L"`) is a fatal error. The reason for this is that there simply isn't any sensible mapping for these special values into integers.

Examples:

```
$foo = pack("WWWW",65,66,67,68);
# foo eq "ABCD"
$foo = pack("W4",65,66,67,68);
# same thing
$foo = pack("W4",0x24b6,0x24b7,0x24b8,0x24b9);
# same thing with Unicode circled letters.
$foo = pack("U4",0x24b6,0x24b7,0x24b8,0x24b9);
# same thing with Unicode circled letters.  You don't get the
# UTF-8 bytes because the U at the start of the format caused
# a switch to U0-mode, so the UTF-8 bytes get joined into
# characters
$foo = pack("C0U4",0x24b6,0x24b7,0x24b8,0x24b9);
# foo eq "\xe2\x92\xb6\xe2\x92\xb7\xe2\x92\xb8\xe2\x92\xb9"
# This is the UTF-8 encoding of the string in the
# previous example

$foo = pack("ccxxcc",65,66,67,68);
# foo eq "AB\0\0CD"

# NOTE: The examples above featuring "W" and "c" are true
# only on ASCII and ASCII-derived systems such as ISO Latin 1
# and UTF-8.  On EBCDIC systems, the first example would be
#      $foo = pack("WWWW",193,194,195,196);

$foo = pack("s2",1,2);
# "\001\000\002\000" on little-endian
# "\000\001\000\002" on big-endian

$foo = pack("a4","abcd","x","y","z");
# "abcd"

$foo = pack("aaaa","abcd","x","y","z");
# "axyz"

$foo = pack("a14","abcdefg");
# "abcdefg\0\0\0\0\0\0\0"

$foo = pack("i9pl", gmtime);
# a real struct tm (on my system anyway)

$utmp_template = "Z8 Z8 Z16 L";
$utmp = pack($utmp_template, @utmp1);
# a struct utmp (BSDish)

@utmp2 = unpack($utmp_template, $utmp);
# "@utmp1" eq "@utmp2"

sub bintodec {
    unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}

$foo = pack('sx2l', 12, 34);
# short 12, two zero bytes padding, long 34
$bar = pack('s@4l', 12, 34);
# short 12, zero fill to position 4, long 34
# $foo eq $bar
$baz = pack('s.l', 12, 4, 34);
# short 12, zero fill to position 4, long 34
```

```
$foo = pack('nN', 42, 4711);
# pack big-endian 16- and 32-bit unsigned integers
$foo = pack('S>L>', 42, 4711);
# exactly the same
$foo = pack('s<l<', -42, 4711);
# pack little-endian 16- and 32-bit signed integers
$foo = pack('(sl)<', -42, 4711);
# exactly the same
```

The same template may generally also be used in `unpack`.

package NAMESPACE
package NAMESPACE VERSION
package NAMESPACE BLOCK
package NAMESPACE VERSION BLOCK

Declares the BLOCK or the rest of the compilation unit as being in the given namespace. The scope of the package declaration is either the supplied code BLOCK or, in the absence of a BLOCK, from the declaration itself through the end of current scope (the enclosing block, file, or `eval`). That is, the forms without a BLOCK are operative through the end of the current scope, just like the `my`, `state`, and `our` operators. All unqualified dynamic identifiers in this scope will be in the given namespace, except where overridden by another `package` declaration or when they're one of the special identifiers that qualify into `main::`, like `STDOUT`, `ARGV`, `ENV`, and the punctuation variables.

A package statement affects dynamic variables only, including those you've used `local` on, but *not* lexically-scoped variables, which are created with `my`, `state`, or `our`. Typically it would be the first declaration in a file included by `require` or `use`. You can switch into a package in more than one place, since this only determines which default symbol table the compiler uses for the rest of that block. You can refer to identifiers in other packages than the current one by prefixing the identifier with the package name and a double colon, as in `$SomePack::var` or `ThatPack::INPUT_HANDLE`. If package name is omitted, the `main` package as assumed. That is, `$::sail` is equivalent to `$main::sail` (as well as to `$main'sail`, still seen in ancient code, mostly from Perl 4).

If VERSION is provided, `package` sets the `$VERSION` variable in the given namespace to a version object with the VERSION provided. VERSION must be a "strict" style version number as defined by the version module: a positive decimal number (integer or decimal-fraction) without exponentiation or else a dotted-decimal v-string with a leading 'v' character and at least three components. You should set `$VERSION` only once per package.

See "Packages" in perlmod for more information about packages, modules, and classes. See perlsub for other scoping issues.

__PACKAGE__

A special token that returns the name of the package in which it occurs.

pipe READHANDLE,WRITEHANDLE

Opens a pair of connected pipes like the corresponding system call. Note that if you set up a loop of piped processes, deadlock can occur unless you are very careful. In addition, note that Perl's pipes use IO buffering, so you may need to set `$|` to flush your WRITEHANDLE after each command, depending on the application.

Returns true on success.

See IPC::Open2, IPC::Open3, and "Bidirectional Communication with Another Process" in perlipc for examples of such things.

On systems that support a close-on-exec flag on files, that flag is set on all newly opened file descriptors whose `filenos` are *higher* than the current value of `$^F` (by default 2 for `STDERR`). See "$^F" in perlvar.

pop ARRAY
pop

Pops and returns the last value of the array, shortening the array by one element.

Returns the undefined value if the array is empty, although this may also happen at other times. If ARRAY is omitted, pops the `@ARGV` array in the main program, but the `@_` array in subroutines, just

like `shift`.

Starting with Perl 5.14, an experimental feature allowed `pop` to take a scalar expression. This experiment has been deemed unsuccessful, and was removed as of Perl 5.24.

pos SCALAR

pos Returns the offset of where the last `m//g` search left off for the variable in question (`$_` is used when the variable is not specified). This offset is in characters unless the (no-longer-recommended) `use bytes` pragma is in effect, in which case the offset is in bytes. Note that 0 is a valid match offset. `undef` indicates that the search position is reset (usually due to match failure, but can also be because no match has yet been run on the scalar).

`pos` directly accesses the location used by the regexp engine to store the offset, so assigning to `pos` will change that offset, and so will also influence the `\G` zero-width assertion in regular expressions. Both of these effects take place for the next match, so you can't affect the position with `pos` during the current match, such as in `(?{pos() = 5})` or `s//pos() = 5/e`.

Setting `pos` also resets the *matched with zero-length* flag, described under "Repeated Patterns Matching a Zero-length Substring" in perlre.

Because a failed `m//gc` match doesn't reset the offset, the return from `pos` won't change either in this case. See perlre and perlop.

print FILEHANDLE LIST

print FILEHANDLE

print LIST

print

Prints a string or a list of strings. Returns true if successful. FILEHANDLE may be a scalar variable containing the name of or a reference to the filehandle, thus introducing one level of indirection. (NOTE: If FILEHANDLE is a variable and the next token is a term, it may be misinterpreted as an operator unless you interpose a `+` or put parentheses around the arguments.) If FILEHANDLE is omitted, prints to the last selected (see `select`) output handle. If LIST is omitted, prints `$_` to the currently selected output handle. To use FILEHANDLE alone to print the content of `$_` to it, you must use a bareword filehandle like `FH`, not an indirect one like `$fh`. To set the default output handle to something other than STDOUT, use the select operation.

The current value of `$,` (if any) is printed between each LIST item. The current value of `$\` (if any) is printed after the entire LIST has been printed. Because print takes a LIST, anything in the LIST is evaluated in list context, including any subroutines whose return lists you pass to `print`. Be careful not to follow the print keyword with a left parenthesis unless you want the corresponding right parenthesis to terminate the arguments to the print; put parentheses around all arguments (or interpose a +, but that doesn't look as good).

If you're storing handles in an array or hash, or in general whenever you're using any expression more complex than a bareword handle or a plain, unsubscripted scalar variable to retrieve it, you will have to use a block returning the filehandle value instead, in which case the LIST may not be omitted:

```
print { $files[$i] } "stuff\n";
print { $OK ? *STDOUT : *STDERR } "stuff\n";
```

Printing to a closed pipe or socket will generate a SIGPIPE signal. See perlipc for more on signal handling.

printf FILEHANDLE FORMAT, LIST

printf FILEHANDLE

printf FORMAT, LIST

printf

Equivalent to `print FILEHANDLE sprintf(FORMAT, LIST)`, except that `$\` (the output record separator) is not appended. The FORMAT and the LIST are actually parsed as a single list. The first argument of the list will be interpreted as the `printf` format. This means that `printf(@_)` will use `$_[0]` as the format. See sprintf for an explanation of the format argument. If `use locale` (including `use locale ':not_characters'`) is in effect and `POSIX::setlocale` has been called, the character used for the decimal separator in formatted floating-point numbers is affected by the `LC_NUMERIC` locale setting. See perllocale and POSIX.

For historical reasons, if you omit the list, `$_` is used as the format; to use FILEHANDLE without a list, you must use a bareword filehandle like `FH`, not an indirect one like `$fh`. However, this will rarely do what you want; if `$_` contains formatting codes, they will be replaced with the empty string and a warning will be emitted if warnings are enabled. Just use `print` if you want to print the contents of `$_`.

Don't fall into the trap of using a `printf` when a simple `print` would do. The `print` is more efficient and less error prone.

prototype FUNCTION
prototype
Returns the prototype of a function as a string (or `undef` if the function has no prototype). FUNCTION is a reference to, or the name of, the function whose prototype you want to retrieve. If FUNCTION is omitted, `$_` is used.

If FUNCTION is a string starting with `CORE::`, the rest is taken as a name for a Perl builtin. If the builtin's arguments cannot be adequately expressed by a prototype (such as `system`), `prototype` returns `undef`, because the builtin does not really behave like a Perl function. Otherwise, the string describing the equivalent prototype is returned.

push ARRAY,LIST
Treats ARRAY as a stack by appending the values of LIST to the end of ARRAY. The length of ARRAY increases by the length of LIST. Has the same effect as

```
for my $value (LIST) {
    $ARRAY[++$#ARRAY] = $value;
}
```

but is more efficient. Returns the number of elements in the array following the completed `push`.

Starting with Perl 5.14, an experimental feature allowed `push` to take a scalar expression. This experiment has been deemed unsuccessful, and was removed as of Perl 5.24.

q/STRING/
qq/STRING/
qw/STRING/
qx/STRING/
Generalized quotes. See "Quote-Like Operators" in perlop.

qr/STRING/
Regexp-like quote. See "Regexp Quote-Like Operators" in perlop.

quotemeta EXPR
quotemeta
Returns the value of EXPR with all the ASCII non-"word" characters backslashed. (That is, all ASCII characters not matching `/[A-Za-z_0-9]/` will be preceded by a backslash in the returned string, regardless of any locale settings.) This is the internal function implementing the `\Q` escape in double-quoted strings. (See below for the behavior on non-ASCII code points.)

If EXPR is omitted, uses `$_`.

quotemeta (and `\Q` ... `\E`) are useful when interpolating strings into regular expressions, because by default an interpolated variable will be considered a mini-regular expression. For example:

```
my $sentence = 'The quick brown fox jumped over the lazy dog';
my $substring = 'quick.*?fox';
$sentence =~ s{$substring}{big bad wolf};
```

Will cause `$sentence` to become `'The big bad wolf jumped over...'`.

On the other hand:

```
my $sentence = 'The quick brown fox jumped over the lazy dog';
my $substring = 'quick.*?fox';
$sentence =~ s{\Q$substring\E}{big bad wolf};
```

Or:

```
my $sentence = 'The quick brown fox jumped over the lazy dog';
my $substring = 'quick.*?fox';
my $quoted_substring = quotemeta($substring);
$sentence =~ s{$quoted_substring}{big bad wolf};
```

Will both leave the sentence as is. Normally, when accepting literal string input from the user, `quotemeta` or `\Q` must be used.

In Perl v5.14, all non-ASCII characters are quoted in non-UTF-8-encoded strings, but not quoted in UTF-8 strings.

Starting in Perl v5.16, Perl adopted a Unicode-defined strategy for quoting non-ASCII characters; the quoting of ASCII characters is unchanged.

Also unchanged is the quoting of non-UTF-8 strings when outside the scope of a `use feature 'unicode_strings'`, which is to quote all characters in the upper Latin1 range. This provides complete backwards compatibility for old programs which do not use Unicode. (Note that `unicode_strings` is automatically enabled within the scope of a `use v5.12` or greater.)

Within the scope of `use locale`, all non-ASCII Latin1 code points are quoted whether the string is encoded as UTF-8 or not. As mentioned above, locale does not affect the quoting of ASCII-range characters. This protects against those locales where characters such as `"|"` are considered to be word characters.

Otherwise, Perl quotes non-ASCII characters using an adaptation from Unicode (see <http://www.unicode.org/reports/tr31/>). The only code points that are quoted are those that have any of the Unicode properties: Pattern_Syntax, Pattern_White_Space, White_Space, Default_Ignorable_Code_Point, or General_Category=Control.

Of these properties, the two important ones are Pattern_Syntax and Pattern_White_Space. They have been set up by Unicode for exactly this purpose of deciding which characters in a regular expression pattern should be quoted. No character that can be in an identifier has these properties.

Perl promises, that if we ever add regular expression pattern metacharacters to the dozen already defined (`\ | ( ) [ { ^ $ * + ? .`), that we will only use ones that have the Pattern_Syntax property. Perl also promises, that if we ever add characters that are considered to be white space in regular expressions (currently mostly affected by `/x`), they will all have the Pattern_White_Space property.

Unicode promises that the set of code points that have these two properties will never change, so something that is not quoted in v5.16 will never need to be quoted in any future Perl release. (Not all the code points that match Pattern_Syntax have actually had characters assigned to them; so there is room to grow, but they are quoted whether assigned or not. Perl, of course, would never use an unassigned code point as an actual metacharacter.)

Quoting characters that have the other 3 properties is done to enhance the readability of the regular expression and not because they actually need to be quoted for regular expression purposes (characters with the White_Space property are likely to be indistinguishable on the page or screen from those with the Pattern_White_Space property; and the other two properties contain non-printing characters).

rand EXPR
rand

Returns a random fractional number greater than or equal to `0` and less than the value of EXPR. (EXPR should be positive.) If EXPR is omitted, the value `1` is used. Currently EXPR with the value `0` is also special-cased as `1` (this was undocumented before Perl 5.8.0 and is subject to change in future versions of Perl). Automatically calls `srand` unless `srand` has already been called. See also `srand`.

Apply `int` to the value returned by `rand` if you want random integers instead of random fractional numbers. For example,

```
int(rand(10))
```

returns a random integer between `0` and `9`, inclusive.

(Note: If your rand function consistently returns numbers that are too large or too small, then your version of Perl was probably compiled with the wrong number of RANDBITS.)

`rand` is not cryptographically secure. You should not rely on it in security-sensitive situations. As of this writing, a number of third-party CPAN modules offer random number generators intended by their authors to be cryptographically secure, including: Data::Entropy, Crypt::Random, Math::Random::Secure, and Math::TrulyRandom.

read FILEHANDLE,SCALAR,LENGTH,OFFSET
read FILEHANDLE,SCALAR,LENGTH
Attempts to read LENGTH *characters* of data into variable SCALAR from the specified FILEHANDLE. Returns the number of characters actually read, `0` at end of file, or undef if there was an error (in the latter case `$!` is also set). SCALAR will be grown or shrunk so that the last character actually read is the last character of the scalar after the read.

An OFFSET may be specified to place the read data at some place in the string other than the beginning. A negative OFFSET specifies placement at that many characters counting backwards from the end of the string. A positive OFFSET greater than the length of SCALAR results in the string being padded to the required size with `"\0"` bytes before the result of the read is appended.

The call is implemented in terms of either Perl's or your system's native *fread*(3) library function. To get a true *read*(2) system call, see sysread.

Note the *characters*: depending on the status of the filehandle, either (8−bit) bytes or characters are read. By default, all filehandles operate on bytes, but for example if the filehandle has been opened with the `:utf8` I/O layer (see `open`, and the open pragma), the I/O will operate on UTF8−encoded Unicode characters, not bytes. Similarly for the `:encoding` layer: in that case pretty much any characters can be read.

readdir DIRHANDLE
Returns the next directory entry for a directory opened by `opendir`. If used in list context, returns all the rest of the entries in the directory. If there are no more entries, returns the undefined value in scalar context and the empty list in list context.

If you're planning to filetest the return values out of a `readdir`, you'd better prepend the directory in question. Otherwise, because we didn't `chdir` there, it would have been testing the wrong file.

```
opendir(my $dh, $some_dir) || die "Can't opendir $some_dir: $!";
my @dots = grep { /^\./ && -f "$some_dir/$_" } readdir($dh);
closedir $dh;
```

As of Perl 5.12 you can use a bare `readdir` in a `while` loop, which will set `$_` on every iteration.

```
opendir(my $dh, $some_dir) || die "Can't open $some_dir: $!";
while (readdir $dh) {
    print "$some_dir/$_\n";
}
closedir $dh;
```

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious failures, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
use 5.012; # so readdir assigns to $_ in a lone while test
```

readline EXPR
readline
Reads from the filehandle whose typeglob is contained in EXPR (or from `*ARGV` if EXPR is not provided). In scalar context, each call reads and returns the next line until end-of-file is reached, whereupon the subsequent call returns `undef`. In list context, reads until end-of-file is reached and returns a list of lines. Note that the notion of "line" used here is whatever you may have defined with `$/` (or `$INPUT_RECORD_SEPARATOR` in English). See "$/" in perlvar.

When `$/` is set to `undef`, when `readline` is in scalar context (i.e., file slurp mode), and when an empty file is read, it returns `''` the first time, followed by `undef` subsequently.

This is the internal function implementing the `<EXPR>` operator, but you can use it directly. The `<EXPR>` operator is discussed in more detail in "I/O Operators" in perlop.

```
my $line = <STDIN>;
my $line = readline(STDIN);     # same thing
```

If `readline` encounters an operating system error, `$!` will be set with the corresponding error message. It can be helpful to check `$!` when you are reading from filehandles you don't trust, such as a tty or a socket. The following example uses the operator form of `readline` and dies if the result is not defined.

```
while ( ! eof($fh) ) {
    defined( $_ = readline $fh ) or die "readline failed: $!";
    ...
}
```

Note that you have can't handle `readline` errors that way with the `ARGV` filehandle. In that case, you have to open each element of `@ARGV` yourself since `eof` handles `ARGV` differently.

```
foreach my $arg (@ARGV) {
    open(my $fh, $arg) or warn "Can't open $arg: $!";

    while ( ! eof($fh) ) {
        defined( $_ = readline $fh )
            or die "readline failed for $arg: $!";
        ...
    }
}
```

readlink EXPR
readlink

Returns the value of a symbolic link, if symbolic links are implemented. If not, raises an exception. If there is a system error, returns the undefined value and sets `$!` (errno). If EXPR is omitted, uses `$_`.

Portability issues: "readlink" in perlport.

readpipe EXPR
readpipe

EXPR is executed as a system command. The collected standard output of the command is returned. In scalar context, it comes back as a single (potentially multi-line) string. In list context, returns a list of lines (however you've defined lines with `$/` (or `$INPUT_RECORD_SEPARATOR` in English)). This is the internal function implementing the `qx/EXPR/` operator, but you can use it directly. The `qx/EXPR/` operator is discussed in more detail in "I/O Operators" in perlop. If EXPR is omitted, uses `$_`.

recv SOCKET,SCALAR,LENGTH,FLAGS

Receives a message on a socket. Attempts to receive LENGTH characters of data into variable SCALAR from the specified SOCKET filehandle. SCALAR will be grown or shrunk to the length actually read. Takes the same flags as the system call of the same name. Returns the address of the sender if SOCKET's protocol supports this; returns an empty string otherwise. If there's an error, returns the undefined value. This call is actually implemented in terms of the *recvfrom*(2) system call. See "UDP: Message Passing" in perlipc for examples.

Note the *characters*: depending on the status of the socket, either (8-bit) bytes or characters are received. By default all sockets operate on bytes, but for example if the socket has been changed using `binmode` to operate with the `:encoding(UTF-8)` I/O layer (see the open pragma), the I/O will operate on UTF8-encoded Unicode characters, not bytes. Similarly for the `:encoding` layer: in that case pretty much any characters can be read.

redo LABEL
redo EXPR
redo

The `redo` command restarts the loop block without evaluating the conditional again. The `continue` block, if any, is not executed. If the LABEL is omitted, the command refers to the innermost enclosing loop. The `redo EXPR` form, available starting in Perl 5.18.0, allows a label name to be computed at run time, and is otherwise identical to `redo LABEL`. Programs that want to lie to themselves about what was just input normally use this command:

```
# a simpleminded Pascal comment stripper
# (warning: assumes no { or } in strings)
LINE: while (<STDIN>) {
    while (s|({.*}.*){.*}|$1 |) {}
    s|{.*}| |;
    if (s|{.*| |) {
        my $front = $_;
        while (<STDIN>) {
            if (/}/) {  # end of comment?
                s|^|$front\{|;
                redo LINE;
            }
        }
    }
    print;
}
```

`redo` cannot be used to retry a block that returns a value such as `eval {}`, `sub {}`, or `do {}`, and should not be used to exit a `grep` or `map` operation.

Note that a block by itself is semantically identical to a loop that executes once. Thus `redo` inside such a block will effectively turn it into a looping construct.

See also `continue` for an illustration of how `last`, `next`, and `redo` work.

Unlike most named operators, this has the same precedence as assignment. It is also exempt from the looks-like-a-function rule, so `redo ("foo")."bar"` will cause "bar" to be part of the argument to `redo`.

ref EXPR

ref Returns a non-empty string if EXPR is a reference, the empty string otherwise. If EXPR is not specified, `$_` will be used. The value returned depends on the type of thing the reference is a reference to.

Builtin types include:

```
SCALAR
ARRAY
HASH
CODE
REF
GLOB
LVALUE
FORMAT
IO
VSTRING
Regexp
```

You can think of `ref` as a `typeof` operator.

```
if (ref($r) eq "HASH") {
    print "r is a reference to a hash.\n";
}
unless (ref($r)) {
    print "r is not a reference at all.\n";
}
```

The return value `LVALUE` indicates a reference to an lvalue that is not a variable. You get this from taking the reference of function calls like `pos` or `substr`. `VSTRING` is returned if the reference points to a version string.

The result `Regexp` indicates that the argument is a regular expression resulting from `qr//`.

If the referenced object has been blessed into a package, then that package name is returned instead. But don't use that, as it's now considered "bad practice". For one reason, an object could be using a

class called `Regexp` or `IO`, or even `HASH`. Also, `ref` doesn't take into account subclasses, like `isa` does.

Instead, use `blessed` (in the Scalar::Util module) for boolean checks, `isa` for specific class checks and `reftype` (also from Scalar::Util) for type checks. (See perlobj for details and a `blessed`/`isa` example.)

See also perlref.

rename OLDNAME,NEWNAME
: Changes the name of a file; an existing file NEWNAME will be clobbered. Returns true for success, false otherwise.

  Behavior of this function varies wildly depending on your system implementation. For example, it will usually not work across file system boundaries, even though the system *mv* command sometimes compensates for this. Other restrictions include whether it works on directories, open files, or pre-existing files. Check perlport and either the *rename*(2) manpage or equivalent system documentation for details.

  For a platform independent `move` function look at the File::Copy module.

  Portability issues: "rename" in perlport.

require VERSION
require EXPR
require
: Demands a version of Perl specified by VERSION, or demands some semantics specified by EXPR or by `$_` if EXPR is not supplied.

  VERSION may be either a numeric argument such as 5.006, which will be compared to `$]`, or a literal of the form v5.6.1, which will be compared to `$^V` (or `$PERL_VERSION` in English). An exception is raised if VERSION is greater than the version of the current Perl interpreter. Compare with `use`, which can do a similar check at compile time.

  Specifying VERSION as a literal of the form v5.6.1 should generally be avoided, because it leads to misleading error messages under earlier versions of Perl that do not support this syntax. The equivalent numeric version should be used instead.

```
require v5.6.1;     # run time version check
require 5.6.1;      # ditto
require 5.006_001;  # ditto; preferred for backwards
                      compatibility
```

  Otherwise, `require` demands that a library file be included if it hasn't already been included. The file is included via the do-FILE mechanism, which is essentially just a variety of `eval` with the caveat that lexical variables in the invoking script will be invisible to the included code. If it were implemented in pure Perl, it would have semantics similar to the following:

```
use Carp 'croak';
use version;

sub require {
    my ($filename) = @_;
    if ( my $version = eval { version->parse($filename) } ) {
        if ( $version > $^V ) {
           my $vn = $version->normal;
           croak "Perl $vn required--this is only $^V, stopped";
        }
        return 1;
    }

    if (exists $INC{$filename}) {
        return 1 if $INC{$filename};
        croak "Compilation failed in require";
    }
```

```
        foreach $prefix (@INC) {
            if (ref($prefix)) {
                #... do other stuff - see text below ....
            }
            # (see text below about possible appending of .pmc
            # suffix to $filename)
            my $realfilename = "$prefix/$filename";
            next if ! -e $realfilename || -d _ || -b _;
            $INC{$filename} = $realfilename;
            my $result = do($realfilename);
                        # but run in caller's namespace

            if (!defined $result) {
                $INC{$filename} = undef;
                croak $@ ? "$@Compilation failed in require"
                         : "Can't locate $filename: $!\n";
            }
            if (!$result) {
                delete $INC{$filename};
                croak "$filename did not return true value";
            }
            $! = 0;
            return $result;
        }
        croak "Can't locate $filename in \@INC ...";
    }
```

Note that the file will not be included twice under the same specified name.

The file must return true as the last statement to indicate successful execution of any initialization code, so it's customary to end such a file with `1;` unless you're sure it'll return true otherwise. But it's better just to put the `1;`, in case you add more statements.

If EXPR is a bareword, `require` assumes a *.pm* extension and replaces `::` with `/` in the filename for you, to make it easy to load standard modules. This form of loading of modules does not risk altering your namespace.

In other words, if you try this:

```
        require Foo::Bar;     # a splendid bareword
```

The require function will actually look for the *Foo/Bar.pm* file in the directories specified in the `@INC` array.

But if you try this:

```
        my $class = 'Foo::Bar';
        require $class;       # $class is not a bareword
    #or
        require "Foo::Bar";   # not a bareword because of the ""
```

The require function will look for the *Foo::Bar* file in the `@INC` array and will complain about not finding *Foo::Bar* there. In this case you can do:

```
        eval "require $class";
```

Now that you understand how `require` looks for files with a bareword argument, there is a little extra functionality going on behind the scenes. Before `require` looks for a *.pm* extension, it will first look for a similar filename with a *.pmc* extension. If this file is found, it will be loaded in place of any file ending in a *.pm* extension.

You can also insert hooks into the import facility by putting Perl code directly into the `@INC` array. There are three forms of hooks: subroutine references, array references, and blessed objects.

Subroutine references are the simplest case. When the inclusion system walks through `@INC` and encounters a subroutine, this subroutine gets called with two parameters, the first a reference to itself, and the second the name of the file to be included (e.g., *Foo/Bar.pm*). The subroutine should return either nothing or else a list of up to four values in the following order:

1. A reference to a scalar, containing any initial source code to prepend to the file or generator output.
2. A filehandle, from which the file will be read.
3. A reference to a subroutine. If there is no filehandle (previous item), then this subroutine is expected to generate one line of source code per call, writing the line into `$_` and returning 1, then finally at end of file returning 0. If there is a filehandle, then the subroutine will be called to act as a simple source filter, with the line as read in `$_`. Again, return 1 for each valid line, and 0 after all lines have been returned.
4. Optional state for the subroutine. The state is passed in as `$_[1]`. A reference to the subroutine itself is passed in as `$_[0]`.

If an empty list, `undef`, or nothing that matches the first 3 values above is returned, then `require` looks at the remaining elements of `@INC`. Note that this filehandle must be a real filehandle (strictly a typeglob or reference to a typeglob, whether blessed or unblessed); tied filehandles will be ignored and processing will stop there.

If the hook is an array reference, its first element must be a subroutine reference. This subroutine is called as above, but the first parameter is the array reference. This lets you indirectly pass arguments to the subroutine.

In other words, you can write:

```
push @INC, \&my_sub;
sub my_sub {
    my ($coderef, $filename) = @_;  # $coderef is \&my_sub
    ...
}
```

or:

```
push @INC, [ \&my_sub, $x, $y, ... ];
sub my_sub {
    my ($arrayref, $filename) = @_;
    # Retrieve $x, $y, ...
    my (undef, @parameters) = @$arrayref;
    ...
}
```

If the hook is an object, it must provide an `INC` method that will be called as above, the first parameter being the object itself. (Note that you must fully qualify the sub's name, as unqualified `INC` is always forced into package `main`.) Here is a typical code layout:

```
# In Foo.pm
package Foo;
sub new { ... }
sub Foo::INC {
    my ($self, $filename) = @_;
    ...
}

# In the main program
push @INC, Foo->new(...);
```

These hooks are also permitted to set the `%INC` entry corresponding to the files they have loaded. See "%INC" in perlvar.

For a yet-more-powerful import facility, see `use` and perlmod.

reset EXPR
reset

Generally used in a `continue` block at the end of a loop to clear variables and reset `m?pattern?` searches so that they work again. The expression is interpreted as a list of single characters (hyphens allowed for ranges). All variables and arrays beginning with one of those letters are reset to their pristine state. If the expression is omitted, one-match searches (`m?pattern?`) are reset to match again. Only resets variables or searches in the current package. Always returns 1. Examples:

```
reset 'X';      # reset all X variables
reset 'a-z';    # reset lower case variables
reset;          # just reset m?one-time? searches
```

Resetting `"A-Z"` is not recommended because you'll wipe out your `@ARGV` and `@INC` arrays and your `%ENV` hash. Resets only package variables; lexical variables are unaffected, but they clean themselves up on scope exit anyway, so you'll probably want to use them instead. See `my`.

return EXPR
return

Returns from a subroutine, `eval`, `do FILE`, `sort` block or regex eval block (but not a `grep` or `map` block) with the value given in EXPR. Evaluation of EXPR may be in list, scalar, or void context, depending on how the return value will be used, and the context may vary from one execution to the next (see `wantarray`). If no EXPR is given, returns an empty list in list context, the undefined value in scalar context, and (of course) nothing at all in void context.

(In the absence of an explicit `return`, a subroutine, `eval`, or `do FILE` automatically returns the value of the last expression evaluated.)

Unlike most named operators, this is also exempt from the looks-like-a-function rule, so `return ("foo")."bar"` will cause `"bar"` to be part of the argument to `return`.

reverse LIST

In list context, returns a list value consisting of the elements of LIST in the opposite order. In scalar context, concatenates the elements of LIST and returns a string value with all characters in the opposite order.

```
print join(", ", reverse "world", "Hello"); # Hello, world

print scalar reverse "dlrow ,", "olleH";    # Hello, world
```

Used without arguments in scalar context, `reverse` reverses `$_`.

```
$_ = "dlrow ,olleH";
print reverse;                      # No output, list context
print scalar reverse;               # Hello, world
```

Note that reversing an array to itself (as in `@a = reverse @a`) will preserve non-existent elements whenever possible; i.e., for non-magical arrays or for tied arrays with `EXISTS` and `DELETE` methods.

This operator is also handy for inverting a hash, although there are some caveats. If a value is duplicated in the original hash, only one of those can be represented as a key in the inverted hash. Also, this has to unwind one hash and build a whole new one, which may take some time on a large hash, such as from a DBM file.

```
my %by_name = reverse %by_address;  # Invert the hash
```

rewinddir DIRHANDLE

Sets the current position to the beginning of the directory for the `readdir` routine on DIRHANDLE.

Portability issues: "rewinddir" in perlport.

rindex STR,SUBSTR,POSITION
rindex STR,SUBSTR

Works just like `index` except that it returns the position of the *last* occurrence of SUBSTR in STR. If POSITION is specified, returns the last occurrence beginning at or before that position.

rmdir FILENAME
rmdir
: Deletes the directory specified by FILENAME if that directory is empty. If it succeeds it returns true; otherwise it returns false and sets `$!` (errno). If FILENAME is omitted, uses `$_`.

  To remove a directory tree recursively (`rm -rf` on Unix) look at the `rmtree` function of the File::Path module.

s///
: The substitution operator. See ''Regexp Quote-Like Operators'' in perlop.

say FILEHANDLE LIST
say FILEHANDLE
say LIST
say
: Just like `print`, but implicitly appends a newline. `say LIST` is simply an abbreviation for `{ local $\ = "\n"; print LIST }`. To use FILEHANDLE without a LIST to print the contents of `$_` to it, you must use a bareword filehandle like `FH`, not an indirect one like `$fh`.

  `say` is available only if the `"say"` feature is enabled or if it is prefixed with `CORE::`. The `"say"` feature is enabled automatically with a `use v5.10` (or higher) declaration in the current scope.

scalar EXPR
: Forces EXPR to be interpreted in scalar context and returns the value of EXPR.

```
my @counts = ( scalar @a, scalar @b, scalar @c );
```

  There is no equivalent operator to force an expression to be interpolated in list context because in practice, this is never needed. If you really wanted to do so, however, you could use the construction `@{[ (some expression) ]}`, but usually a simple `(some expression)` suffices.

  Because `scalar` is a unary operator, if you accidentally use a parenthesized list for the EXPR, this behaves as a scalar comma expression, evaluating all but the last element in void context and returning the final element evaluated in scalar context. This is seldom what you want.

  The following single statement:

```
print uc(scalar(foo(), $bar)), $baz;
```

  is the moral equivalent of these two:

```
foo();
print(uc($bar), $baz);
```

  See perlop for more details on unary operators and the comma operator, and perldata for details on evaluating a hash in scalar contex.

seek FILEHANDLE,POSITION,WHENCE
: Sets FILEHANDLE's position, just like the *fseek*(3) call of C `stdio`. FILEHANDLE may be an expression whose value gives the name of the filehandle. The values for WHENCE are `0` to set the new position *in bytes* to POSITION; `1` to set it to the current position plus POSITION; and `2` to set it to EOF plus POSITION, typically negative. For WHENCE you may use the constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` (start of the file, current position, end of the file) from the Fcntl module. Returns `1` on success, false otherwise.

  Note the emphasis on bytes: even if the filehandle has been set to operate on characters (for example using the `:encoding(UTF-8)` I/O layer), the `seek`, `tell`, and `sysseek` family of functions use byte offsets, not character offsets, because seeking to a character offset would be very slow in a UTF-8 file.

  If you want to position the file for `sysread` or `syswrite`, don't use `seek`, because buffering makes its effect on the file's read-write position unpredictable and non-portable. Use `sysseek` instead.

  Due to the rules and rigors of ANSI C, on some systems you have to do a seek whenever you switch between reading and writing. Amongst other things, this may have the effect of calling stdio's *clearerr*(3). A WHENCE of `1` (`SEEK_CUR`) is useful for not moving the file position:

```
seek($fh, 0, 1);
```

  This is also useful for applications emulating `tail -f`. Once you hit EOF on your read and then

sleep for a while, you (probably) have to stick in a dummy `seek` to reset things. The `seek` doesn't change the position, but it *does* clear the end-of-file condition on the handle, so that the next `readline FILE` makes Perl try again to read something. (We hope.)

If that doesn't work (some I/O implementations are particularly cantankerous), you might need something like this:

```
for (;;) {
    for ($curpos = tell($fh); $_ = readline($fh);
         $curpos = tell($fh)) {
        # search for some stuff and put it into files
    }
    sleep($for_a_while);
    seek($fh, $curpos, 0);
}
```

seekdir DIRHANDLE,POS
: Sets the current position for the `readdir` routine on DIRHANDLE. POS must be a value returned by `telldir`. `seekdir` also has the same caveats about possible directory compaction as the corresponding system library routine.

select FILEHANDLE
select
: Returns the currently selected filehandle. If FILEHANDLE is supplied, sets the new current default filehandle for output. This has two effects: first, a `write` or a `print` without a filehandle default to this FILEHANDLE. Second, references to variables related to output will refer to this output channel.

  For example, to set the top-of-form format for more than one output channel, you might do the following:

```
select(REPORT1);
$^ = 'report1_top';
select(REPORT2);
$^ = 'report2_top';
```

  FILEHANDLE may be an expression whose value gives the name of the actual filehandle. Thus:

```
my $oldfh = select(STDERR); $| = 1; select($oldfh);
```

  Some programmers may prefer to think of filehandles as objects with methods, preferring to write the last example as:

```
STDERR->autoflush(1);
```

  (Prior to Perl version 5.14, you have to `use IO::Handle;` explicitly first.)

  Portability issues: "select" in perlport.

select RBITS,WBITS,EBITS,TIMEOUT
: This calls the *select*(2) syscall with the bit masks specified, which can be constructed using `fileno` and `vec`, along these lines:

```
my $rin = my $win = my $ein = '';
vec($rin, fileno(STDIN),  1) = 1;
vec($win, fileno(STDOUT), 1) = 1;
$ein = $rin | $win;
```

  If you want to select on many filehandles, you may wish to write a subroutine like this:

```
    sub fhbits {
        my @fhlist = @_;
        my $bits = "";
        for my $fh (@fhlist) {
            vec($bits, fileno($fh), 1) = 1;
        }
        return $bits;
    }
    my $rin = fhbits(\*STDIN, $tty, $mysock);
```

The usual idiom is:

```
 my ($nfound, $timeleft) =
   select(my $rout = $rin, my $wout = $win, my $eout = $ein,
                                                        $timeout);
```

or to block until something becomes ready just do this

```
 my $nfound =
   select(my $rout = $rin, my $wout = $win, my $eout = $ein, undef);
```

Most systems do not bother to return anything useful in `$timeleft`, so calling `select` in scalar context just returns `$nfound`.

Any of the bit masks can also be `undef`. The timeout, if specified, is in seconds, which may be fractional. Note: not all implementations are capable of returning the `$timeleft`. If not, they always return `$timeleft` equal to the supplied `$timeout`.

You can effect a sleep of 250 milliseconds this way:

```
    select(undef, undef, undef, 0.25);
```

Note that whether `select` gets restarted after signals (say, SIGALRM) is implementation-dependent. See also perlport for notes on the portability of `select`.

On error, `select` behaves just like *select*(2): it returns `-1` and sets `$!`.

On some Unixes, *select*(2) may report a socket file descriptor as "ready for reading" even when no data is available, and thus any subsequent `read` would block. This can be avoided if you always use `O_NONBLOCK` on the socket. See *select*(2) and *fcntl*(2) for further details.

The standard `IO::Select` module provides a user-friendlier interface to `select`, mostly because it does all the bit-mask work for you.

WARNING: One should not attempt to mix buffered I/O (like `read` or `readline`) with `select`, except as permitted by POSIX, and even then only on POSIX systems. You have to use `sysread` instead.

Portability issues: "select" in perlport.

semctl ID,SEMNUM,CMD,ARG
: Calls the System V IPC function *semctl*(2). You'll probably have to say

```
    use IPC::SysV;
```

first to get the correct constant definitions. If CMD is IPC_STAT or GETALL, then ARG must be a variable that will hold the returned semid_ds structure or semaphore value array. Returns like `ioctl`: the undefined value for error, `"0 but true"` for zero, or the actual return value otherwise. The ARG must consist of a vector of native short integers, which may be created with `pack("s!",(0)x$nsem)`. See also "SysV IPC" in perlipc and the documentation for `IPC::SysV` and `IPC::Semaphore`.

Portability issues: "semctl" in perlport.

semget KEY,NSEMS,FLAGS
: Calls the System V IPC function *semget*(2). Returns the semaphore id, or the undefined value on error. See also "SysV IPC" in perlipc and the documentation for `IPC::SysV` and `IPC::Semaphore`.

Portability issues: "semget" in perlport.

semop KEY,OPSTRING
: Calls the System V IPC function *semop*(2) for semaphore operations such as signalling and waiting. OPSTRING must be a packed array of semop structures. Each semop structure can be generated with `pack("s!3", $semnum, $semop, $semflag)`. The length of OPSTRING implies the number of semaphore operations. Returns true if successful, false on error. As an example, the following code waits on semaphore `$semnum` of semaphore id `$semid`:

```
my $semop = pack("s!3", $semnum, -1, 0);
die "Semaphore trouble: $!\n" unless semop($semid, $semop);
```

To signal the semaphore, replace `-1` with `1`. See also "SysV IPC" in perlipc and the documentation for `IPC::SysV` and `IPC::Semaphore`.

Portability issues: "semop" in perlport.

send SOCKET,MSG,FLAGS,TO
send SOCKET,MSG,FLAGS
: Sends a message on a socket. Attempts to send the scalar MSG to the SOCKET filehandle. Takes the same flags as the system call of the same name. On unconnected sockets, you must specify a destination to *send to*, in which case it does a *sendto*(2) syscall. Returns the number of characters sent, or the undefined value on error. The *sendmsg*(2) syscall is currently unimplemented. See "UDP: Message Passing" in perlipc for examples.

Note the *characters*: depending on the status of the socket, either (8-bit) bytes or characters are sent. By default all sockets operate on bytes, but for example if the socket has been changed using `binmode` to operate with the `:encoding(UTF-8)` I/O layer (see `open`, or the open pragma), the I/O will operate on UTF-8 encoded Unicode characters, not bytes. Similarly for the `:encoding` layer: in that case pretty much any characters can be sent.

setpgrp PID,PGRP
: Sets the current process group for the specified PID, `0` for the current process. Raises an exception when used on a machine that doesn't implement POSIX *setpgid*(2) or BSD *setpgrp*(2). If the arguments are omitted, it defaults to `0,0`. Note that the BSD 4.2 version of `setpgrp` does not accept any arguments, so only `setpgrp(0,0)` is portable. See also `POSIX::setsid()`.

Portability issues: "setpgrp" in perlport.

setpriority WHICH,WHO,PRIORITY
: Sets the current priority for a process, a process group, or a user. (See *setpriority*(2).) Raises an exception when used on a machine that doesn't implement *setpriority*(2).

Portability issues: "setpriority" in perlport.

setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL
: Sets the socket option requested. Returns `undef` on error. Use integer constants provided by the `Socket` module for LEVEL and OPNAME. Values for LEVEL can also be obtained from getprotobyname. OPTVAL might either be a packed string or an integer. An integer OPTVAL is shorthand for pack("i", OPTVAL).

An example disabling Nagle's algorithm on a socket:

```
use Socket qw(IPPROTO_TCP TCP_NODELAY);
setsockopt($socket, IPPROTO_TCP, TCP_NODELAY, 1);
```

Portability issues: "setsockopt" in perlport.

shift ARRAY
shift
: Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If there are no elements in the array, returns the undefined value. If ARRAY is omitted, shifts the `@_` array within the lexical scope of subroutines and formats, and the `@ARGV` array outside a subroutine and also within the lexical scopes established by the `eval STRING`, `BEGIN {}`, `INIT {}`, `CHECK {}`, `UNITCHECK {}`, and `END {}` constructs.

Starting with Perl 5.14, an experimental feature allowed `shift` to take a scalar expression. This

experiment has been deemed unsuccessful, and was removed as of Perl 5.24.

See also `unshift`, `push`, and `pop`. `shift` and `unshift` do the same thing to the left end of an array that `pop` and `push` do to the right end.

shmctl ID,CMD,ARG
Calls the System V IPC function shmctl. You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If CMD is `IPC_STAT`, then ARG must be a variable that will hold the returned `shmid_ds` structure. Returns like ioctl: `undef` for error; "`0` but true" for zero; and the actual return value otherwise. See also "SysV IPC" in perlipc and the documentation for `IPC::SysV`.

Portability issues: "shmctl" in perlport.

shmget KEY,SIZE,FLAGS
Calls the System V IPC function shmget. Returns the shared memory segment id, or `undef` on error. See also "SysV IPC" in perlipc and the documentation for `IPC::SysV`.

Portability issues: "shmget" in perlport.

shmread ID,VAR,POS,SIZE
shmwrite ID,STRING,POS,SIZE
Reads or writes the System V shared memory segment ID starting at position POS for size SIZE by attaching to it, copying in/out, and detaching from it. When reading, VAR must be a variable that will hold the data read. When writing, if STRING is too long, only SIZE bytes are used; if STRING is too short, nulls are written to fill out SIZE bytes. Return true if successful, false on error. `shmread` taints the variable. See also "SysV IPC" in perlipc and the documentation for `IPC::SysV` and the `IPC::Shareable` module from CPAN.

Portability issues: "shmread" in perlport and "shmwrite" in perlport.

shutdown SOCKET,HOW
Shuts down a socket connection in the manner indicated by HOW, which has the same interpretation as in the syscall of the same name.

```
shutdown($socket, 0);    # I/we have stopped reading data
shutdown($socket, 1);    # I/we have stopped writing data
shutdown($socket, 2);    # I/we have stopped using this socket
```

This is useful with sockets when you want to tell the other side you're done writing but not done reading, or vice versa. It's also a more insistent form of close because it also disables the file descriptor in any forked copies in other processes.

Returns `1` for success; on error, returns `undef` if the first argument is not a valid filehandle, or returns `0` and sets `$!` for any other failure.

sin EXPR
sin Returns the sine of EXPR (expressed in radians). If EXPR is omitted, returns sine of `$_`.

For the inverse sine operation, you may use the `Math::Trig::asin` function, or use this relation:

```
sub asin { atan2($_[0], sqrt(1 - $_[0] * $_[0])) }
```

sleep EXPR
sleep
Causes the script to sleep for (integer) EXPR seconds, or forever if no argument is given. Returns the integer number of seconds actually slept.

May be interrupted if the process receives a signal such as `SIGALRM`.

```
eval {
    local $SIG{ALRM} = sub { die "Alarm!\n" };
    sleep;
};
die $@ unless $@ eq "Alarm!\n";
```

You probably cannot mix `alarm` and `sleep` calls, because `sleep` is often implemented using `alarm`.

On some older systems, it may sleep up to a full second less than what you requested, depending on how it counts seconds. Most modern systems always sleep the full amount. They may appear to sleep longer than that, however, because your process might not be scheduled right away in a busy multitasking system.

For delays of finer granularity than one second, the Time::HiRes module (from CPAN, and starting from Perl 5.8 part of the standard distribution) provides `usleep`. You may also use Perl's four-argument version of `select` leaving the first three arguments undefined, or you might be able to use the `syscall` interface to access *setitimer*(2) if your system supports it. See perlfaq8 for details.

See also the POSIX module's `pause` function.

socket SOCKET,DOMAIN,TYPE,PROTOCOL

Opens a socket of the specified kind and attaches it to filehandle SOCKET. DOMAIN, TYPE, and PROTOCOL are specified the same as for the syscall of the same name. You should `use Socket` first to get the proper definitions imported. See the examples in "Sockets: Client/Server Communication" in perlipc.

On systems that support a close-on-exec flag on files, the flag will be set for the newly opened file descriptor, as determined by the value of `$^F`. See "$^F" in perlvar.

socketpair SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL

Creates an unnamed pair of sockets in the specified domain, of the specified type. DOMAIN, TYPE, and PROTOCOL are specified the same as for the syscall of the same name. If unimplemented, raises an exception. Returns true if successful.

On systems that support a close-on-exec flag on files, the flag will be set for the newly opened file descriptors, as determined by the value of `$^F`. See "$^F" in perlvar.

Some systems define `pipe` in terms of `socketpair`, in which a call to `pipe($rdr, $wtr)` is essentially:

```
use Socket;
socketpair(my $rdr, my $wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC);
shutdown($rdr, 1);        # no more writing for reader
shutdown($wtr, 0);        # no more reading for writer
```

See perlipc for an example of socketpair use. Perl 5.8 and later will emulate socketpair using IP sockets to localhost if your system implements sockets but not socketpair.

Portability issues: "socketpair" in perlport.

sort SUBNAME LIST
sort BLOCK LIST
sort LIST

In list context, this sorts the LIST and returns the sorted list value. In scalar context, the behaviour of `sort` is undefined.

If SUBNAME or BLOCK is omitted, `sorts` in standard string comparison order. If SUBNAME is specified, it gives the name of a subroutine that returns an integer less than, equal to, or greater than `0`, depending on how the elements of the list are to be ordered. (The `<=>` and `cmp` operators are extremely useful in such routines.) SUBNAME may be a scalar variable name (unsubscripted), in which case the value provides the name of (or a reference to) the actual subroutine to use. In place of a SUBNAME, you can provide a BLOCK as an anonymous, in-line sort subroutine.

If the subroutine's prototype is `($$)`, the elements to be compared are passed by reference in `@_`, as for a normal subroutine. This is slower than unprototyped subroutines, where the elements to be compared are passed into the subroutine as the package global variables `$a` and `$b` (see example below).

If the subroutine is an XSUB, the elements to be compared are pushed on to the stack, the way arguments are usually passed to XSUBs. `$a` and `$b` are not set.

The values to be compared are always passed by reference and should not be modified.

You also cannot exit out of the sort block or subroutine using any of the loop control operators described in perlsyn or with `goto`.

When `use locale` (but not `use locale ':not_characters'`) is in effect, `sort LIST` sorts LIST according to the current collation locale. See perllocale.

`sort` returns aliases into the original list, much as a for loop's index variable aliases the list elements. That is, modifying an element of a list returned by `sort` (for example, in a `foreach`, `map` or `grep`) actually modifies the element in the original list. This is usually something to be avoided when writing clear code.

Perl 5.6 and earlier used a quicksort algorithm to implement sort. That algorithm was not stable and *could* go quadratic. (A *stable* sort preserves the input order of elements that compare equal. Although quicksort's run time is O(NlogN) when averaged over all arrays of length N, the time can be O(N**2), *quadratic* behavior, for some inputs.) In 5.7, the quicksort implementation was replaced with a stable mergesort algorithm whose worst-case behavior is O(NlogN). But benchmarks indicated that for some inputs, on some platforms, the original quicksort was faster. 5.8 has a sort pragma for limited control of the sort. Its rather blunt control of the underlying algorithm may not persist into future Perls, but the ability to characterize the input or output in implementation independent ways quite probably will.

Examples:

```
# sort lexically
my @articles = sort @files;

# same thing, but with explicit sort routine
my @articles = sort {$a cmp $b} @files;

# now case-insensitively
my @articles = sort {fc($a) cmp fc($b)} @files;

# same thing in reversed order
my @articles = sort {$b cmp $a} @files;

# sort numerically ascending
my @articles = sort {$a <=> $b} @files;

# sort numerically descending
my @articles = sort {$b <=> $a} @files;

# this sorts the %age hash by value instead of key
# using an in-line function
my @eldest = sort { $age{$b} <=> $age{$a} } keys %age;

# sort using explicit subroutine name
sub byage {
    $age{$a} <=> $age{$b};  # presuming numeric
}
my @sortedclass = sort byage @class;

sub backwards { $b cmp $a }
my @harry  = qw(dog cat x Cain Abel);
my @george = qw(gone chased yz Punished Axed);
print sort @harry;
    # prints AbelCaincatdogx
print sort backwards @harry;
    # prints xdogcatCainAbel
print sort @george, 'to', @harry;
    # prints AbelAxedCainPunishedcatchaseddoggonetoxyz

# inefficiently sort by descending numeric compare using
```

```
    # the first integer after the first = sign, or the
    # whole record case-insensitively otherwise

    my @new = sort {
        ($b =~ /=(\d+)/)[0] <=> ($a =~ /=(\d+)/)[0]
                            ||
                    fc($a)  cmp  fc($b)
    } @old;

    # same thing, but much more efficiently;
    # we'll build auxiliary indices instead
    # for speed
    my (@nums, @caps);
    for (@old) {
        push @nums, ( /=(\d+)/ ? $1 : undef );
        push @caps, fc($_);
    }

    my @new = @old[ sort {
                          $nums[$b] <=> $nums[$a]
                                    ||
                          $caps[$a] cmp $caps[$b]
                        } 0..$#old
                  ];

    # same thing, but without any temps
    my @new = map { $_->[0] }
           sort { $b->[1] <=> $a->[1]
                            ||
                  $a->[2] cmp $b->[2]
           } map { [$_, /=(\d+)/, fc($_)] } @old;

    # using a prototype allows you to use any comparison subroutine
    # as a sort subroutine (including other package's subroutines)
    package Other;
    sub backwards ($$) { $_[1] cmp $_[0]; }  # $a and $b are
                                             # not set here
    package main;
    my @new = sort Other::backwards @old;

    # guarantee stability, regardless of algorithm
    use sort 'stable';
    my @new = sort { substr($a, 3, 5) cmp substr($b, 3, 5) } @old;

    # force use of mergesort (not portable outside Perl 5.8)
    use sort '_mergesort';  # note discouraging _
    my @new = sort { substr($a, 3, 5) cmp substr($b, 3, 5) } @old;
```

Warning: syntactical care is required when sorting the list returned from a function. If you want to sort the list returned by the function call `find_records(@key)`, you can use:

```
    my @contact = sort { $a cmp $b } find_records @key;
    my @contact = sort +find_records(@key);
    my @contact = sort &find_records(@key);
    my @contact = sort(find_records(@key));
```

If instead you want to sort the array `@key` with the comparison routine `find_records()` then you can use:

```
    my @contact = sort { find_records() } @key;
    my @contact = sort find_records(@key);
    my @contact = sort(find_records @key);
    my @contact = sort(find_records (@key));
```

`$a` and `$b` are set as package globals in the package the *sort()* is called from. That means `$main::a` and `$main::b` (or `$::a` and `$::b`) in the `main` package, `$FooPack::a` and `$FooPack::b` in the `FooPack` package, etc. If the sort block is in scope of a `my` or `state` declaration of `$a` and/or `$b`, you *must* spell out the full name of the variables in the sort block :

```
package main;
my $a = "C"; # DANGER, Will Robinson, DANGER !!!

print sort { $a cmp $b }               qw(A C E G B D F H);
                                       # WRONG
sub badlexi { $a cmp $b }
print sort badlexi                     qw(A C E G B D F H);
                                       # WRONG
# the above prints BACFEDGH or some other incorrect ordering

print sort { $::a cmp $::b }           qw(A C E G B D F H);
                                       # OK
print sort { our $a cmp our $b }       qw(A C E G B D F H);
                                       # also OK
print sort { our ($a, $b); $a cmp $b } qw(A C E G B D F H);
                                       # also OK
sub lexi { our $a cmp our $b }
print sort lexi                        qw(A C E G B D F H);
                                       # also OK
# the above print ABCDEFGH
```

With proper care you may mix package and my (or state) `$a` and/or `$b`:

```
my $a = {
   tiny   => -2,
   small  => -1,
   normal => 0,
   big    => 1,
   huge   => 2
};

say sort { $a->{our $a} <=> $a->{our $b} }
    qw{ huge normal tiny small big};

# prints tinysmallnormalbighuge
```

`$a` and `$b` are implicitely local to the *sort()* execution and regain their former values upon completing the sort.

Sort subroutines written using `$a` and `$b` are bound to their calling package. It is possible, but of limited interest, to define them in a different package, since the subroutine must still refer to the calling package's `$a` and `$b` :

```
package Foo;
sub lexi { $Bar::a cmp $Bar::b }
package Bar;
... sort Foo::lexi ...
```

Use the prototyped versions (see above) for a more generic alternative.

The comparison function is required to behave. If it returns inconsistent results (sometimes saying `$x[1]` is less than `$x[2]` and sometimes saying the opposite, for example) the results are not well-defined.

Because <=> returns `undef` when either operand is `NaN` (not-a-number), be careful when sorting with a comparison function like `$a <=> $b` any lists that might contain a `NaN`. The following example takes advantage that `NaN != NaN` to eliminate any `NaN`s from the input list.

```
my @result = sort { $a <=> $b } grep { $_ == $_ } @input;
```

splice ARRAY,OFFSET,LENGTH,LIST
splice ARRAY,OFFSET,LENGTH
splice ARRAY,OFFSET
splice ARRAY

Removes the elements designated by OFFSET and LENGTH from an array, and replaces them with the elements of LIST, if any. In list context, returns the elements removed from the array. In scalar context, returns the last element removed, or `undef` if no elements are removed. The array grows or shrinks as necessary. If OFFSET is negative then it starts that far from the end of the array. If LENGTH is omitted, removes everything from OFFSET onward. If LENGTH is negative, removes the elements from OFFSET onward except for −LENGTH elements at the end of the array. If both OFFSET and LENGTH are omitted, removes everything. If OFFSET is past the end of the array and a LENGTH was provided, Perl issues a warning, and splices at the end of the array.

The following equivalences hold (assuming `$#a >= $i`)

```
push(@a,$x,$y)      splice(@a,@a,0,$x,$y)
pop(@a)             splice(@a,-1)
shift(@a)           splice(@a,0,1)
unshift(@a,$x,$y)   splice(@a,0,0,$x,$y)
$a[$i] = $y         splice(@a,$i,1,$y)
```

`splice` can be used, for example, to implement n−ary queue processing:

```
sub nary_print {
  my $n = shift;
  while (my @next_n = splice @_, 0, $n) {
    say join q{ -- }, @next_n;
  }
}

nary_print(3, qw(a b c d e f g h));
# prints:
#   a -- b -- c
#   d -- e -- f
#   g -- h
```

Starting with Perl 5.14, an experimental feature allowed `splice` to take a scalar expression. This experiment has been deemed unsuccessful, and was removed as of Perl 5.24.

split /PATTERN/,EXPR,LIMIT
split /PATTERN/,EXPR
split /PATTERN/
split

Splits the string EXPR into a list of strings and returns the list in list context, or the size of the list in scalar context. (Prior to Perl 5.11, it also overwrote `@_` with the list in void and scalar context. If you target old perls, beware.)

If only PATTERN is given, EXPR defaults to `$_`.

Anything in EXPR that matches PATTERN is taken to be a separator that separates the EXPR into substrings (called "*fields*") that do **not** include the separator. Note that a separator may be longer than one character or even have no characters at all (the empty string, which is a zero-width match).

The PATTERN need not be constant; an expression may be used to specify a pattern that varies at runtime.

If PATTERN matches the empty string, the EXPR is split at the match position (between characters). As an example, the following:

```
print join(':', split(/b/, 'abc')), "\n";
```

uses the `b` in `'abc'` as a separator to produce the output `a:c`. However, this:

```
print join(':', split(//, 'abc')), "\n";
```

uses empty string matches as separators to produce the output `a:b:c`; thus, the empty string may be used to split EXPR into a list of its component characters.

As a special case for `split`, the empty pattern given in match operator syntax (`//`) specifically matches the empty string, which is contrary to its usual interpretation as the last successful match.

If PATTERN is `/^/`, then it is treated as if it used the multiline modifier (`/^/m`), since it isn't much use otherwise.

`/m` and any of the other pattern modifiers valid for `qr` (summarized in "qr/STRING/msixpodualn" in perlop) may be specified explicitly.

As another special case, `split` emulates the default behavior of the command line tool **awk** when the PATTERN is either omitted or a string composed of a single space character (such as `' '` or `"\x20"`, but not e.g. `/ /`). In this case, any leading whitespace in EXPR is removed before splitting occurs, and the PATTERN is instead treated as if it were `/\s+/`; in particular, this means that *any* contiguous whitespace (not just a single space character) is used as a separator. However, this special treatment can be avoided by specifying the pattern `/ /` instead of the string `" "`, thereby allowing only a single space character to be a separator. In earlier Perls this special case was restricted to the use of a plain `" "` as the pattern argument to split; in Perl 5.18.0 and later this special case is triggered by any expression which evaluates to the simple string `" "`.

If omitted, PATTERN defaults to a single space, `" "`, triggering the previously described *awk* emulation.

If LIMIT is specified and positive, it represents the maximum number of fields into which the EXPR may be split; in other words, LIMIT is one greater than the maximum number of times EXPR may be split. Thus, the LIMIT value `1` means that EXPR may be split a maximum of zero times, producing a maximum of one field (namely, the entire value of EXPR). For instance:

```
print join(':', split(//, 'abc', 1)), "\n";
```

produces the output `abc`, and this:

```
print join(':', split(//, 'abc', 2)), "\n";
```

produces the output `a:bc`, and each of these:

```
print join(':', split(//, 'abc', 3)), "\n";
print join(':', split(//, 'abc', 4)), "\n";
```

produces the output `a:b:c`.

If LIMIT is negative, it is treated as if it were instead arbitrarily large; as many fields as possible are produced.

If LIMIT is omitted (or, equivalently, zero), then it is usually treated as if it were instead negative but with the exception that trailing empty fields are stripped (empty leading fields are always preserved); if all fields are empty, then all fields are considered to be trailing (and are thus stripped in this case). Thus, the following:

```
print join(':', split(/,/, 'a,b,c,,,')), "\n";
```

produces the output `a:b:c`, but the following:

```
print join(':', split(/,/, 'a,b,c,,,', -1)), "\n";
```

produces the output `a:b:c:::`.

In time-critical applications, it is worthwhile to avoid splitting into more fields than necessary. Thus, when assigning to a list, if LIMIT is omitted (or zero), then LIMIT is treated as though it were one larger than the number of variables in the list; for the following, LIMIT is implicitly 3:

```
my ($login, $passwd) = split(/:/);
```

Note that splitting an EXPR that evaluates to the empty string always produces zero fields, regardless of the LIMIT specified.

An empty leading field is produced when there is a positive-width match at the beginning of EXPR. For instance:

```
print join(':', split(/ /, ' abc')), "\n";
```

produces the output `:abc`. However, a zero-width match at the beginning of EXPR never produces an empty field, so that:

```
print join(':', split(//, ' abc'));
```

produces the output `:a:b:c` (rather than `: :a:b:c`).

An empty trailing field, on the other hand, is produced when there is a match at the end of EXPR, regardless of the length of the match (of course, unless a non-zero LIMIT is given explicitly, such fields are removed, as in the last example). Thus:

```
print join(':', split(//, ' abc', -1)), "\n";
```

produces the output `:a:b:c:`.

If the PATTERN contains capturing groups, then for each separator, an additional field is produced for each substring captured by a group (in the order in which the groups are specified, as per backreferences); if any group does not match, then it captures the `undef` value instead of a substring. Also, note that any such additional field is produced whenever there is a separator (that is, whenever a split occurs), and such an additional field does **not** count towards the LIMIT. Consider the following expressions evaluated in list context (each returned list is provided in the associated comment):

```
split(/-|,/, "1-10,20", 3)
# ('1', '10', '20')

split(/(-|,)/, "1-10,20", 3)
# ('1', '-', '10', ',', '20')

split(/-|(,)/, "1-10,20", 3)
# ('1', undef, '10', ',', '20')

split(/(-)|,/, "1-10,20", 3)
# ('1', '-', '10', undef, '20')

split(/(-)|(,)/, "1-10,20", 3)
# ('1', '-', undef, '10', undef, ',', '20')
```

sprintf FORMAT, LIST

Returns a string formatted by the usual `printf` conventions of the C library function `sprintf`. See below for more details and see *sprintf*(3) or *printf*(3) on your system for an explanation of the general principles.

For example:

```
# Format number with up to 8 leading zeroes
my $result = sprintf("%08d", $number);

# Round number to 3 digits after decimal point
my $rounded = sprintf("%.3f", $number);
```

Perl does its own `sprintf` formatting: it emulates the C function *sprintf*(3), but doesn't use it except for floating-point numbers, and even then only standard modifiers are allowed. Non-standard extensions in your local *sprintf*(3) are therefore unavailable from Perl.

Unlike `printf`, `sprintf` does not do what you probably mean when you pass it an array as your first argument. The array is given scalar context, and instead of using the 0th element of the array as the format, Perl will use the count of elements in the array as the format, which is almost never useful.

Perl's `sprintf` permits the following universally-known conversions:

```
%%    a percent sign
%c    a character with the given number
%s    a string
%d    a signed integer, in decimal
%u    an unsigned integer, in decimal
%o    an unsigned integer, in octal
%x    an unsigned integer, in hexadecimal
%e    a floating-point number, in scientific notation
%f    a floating-point number, in fixed decimal notation
%g    a floating-point number, in %e or %f notation
```

In addition, Perl permits the following widely-supported conversions:

```
%X    like %x, but using upper-case letters
%E    like %e, but using an upper-case "E"
%G    like %g, but with an upper-case "E" (if applicable)
%b    an unsigned integer, in binary
%B    like %b, but using an upper-case "B" with the # flag
%p    a pointer (outputs the Perl value's address in hexadecimal)
%n    special: *stores* the number of characters output so far
      into the next argument in the parameter list
%a    hexadecimal floating point
%A    like %a, but using upper-case letters
```

Finally, for backward (and we do mean "backward") compatibility, Perl permits these unnecessary but widely-supported conversions:

```
%i    a synonym for %d
%D    a synonym for %ld
%U    a synonym for %lu
%O    a synonym for %lo
%F    a synonym for %f
```

Note that the number of exponent digits in the scientific notation produced by `%e`, `%E`, `%g` and `%G` for numbers with the modulus of the exponent less than 100 is system-dependent: it may be three or less (zero-padded as necessary). In other words, 1.23 times ten to the 99th may be either "1.23e99" or "1.23e099". Similarly for `%a` and `%A`: the exponent or the hexadecimal digits may float: especially the "long doubles" Perl configuration option may cause surprises.

Between the `%` and the format letter, you may specify several additional attributes controlling the interpretation of the format. In order, these are:

format parameter index
: An explicit format parameter index, such as `2$`. By default sprintf will format the next unused argument in the list, but this allows you to take the arguments out of order:

```
printf '%2$d %1$d', 12, 34;      # prints "34 12"
printf '%3$d %d %1$d', 1, 2, 3;  # prints "3 1 1"
```

flags
: one or more of:

```
space   prefix non-negative number with a space
+       prefix non-negative number with a plus sign
-       left-justify within the field
0       use zeros, not spaces, to right-justify
#       ensure the leading "0" for any octal,
        prefix non-zero hexadecimal with "0x" or "0X",
        prefix non-zero binary with "0b" or "0B"
```

For example:

```
printf '<% d>',  12;   # prints "< 12>"
printf '<% d>',   0;   # prints "< 0>"
printf '<% d>', -12;   # prints "<-12>"
printf '<%+d>',  12;   # prints "<+12>"
printf '<%+d>',   0;   # prints "<+0>"
printf '<%+d>', -12;   # prints "<-12>"
printf '<%6s>',  12;   # prints "<    12>"
printf '<%-6s>', 12;   # prints "<12    >"
printf '<%06s>', 12;   # prints "<000012>"
printf '<%#o>',  12;   # prints "<014>"
printf '<%#x>',  12;   # prints "<0xc>"
printf '<%#X>',  12;   # prints "<0XC>"
printf '<%#b>',  12;   # prints "<0b1100>"
printf '<%#B>',  12;   # prints "<0B1100>"
```

When a space and a plus sign are given as the flags at once, the space is ignored.

```
printf '<%+ d>', 12;   # prints "<+12>"
printf '<% +d>', 12;   # prints "<+12>"
```

When the # flag and a precision are given in the `%o` conversion, the precision is incremented if it's necessary for the leading "0".

```
printf '<%#.5o>', 012;      # prints "<00012>"
printf '<%#.5o>', 012345;   # prints "<012345>"
printf '<%#.0o>', 0;        # prints "<0>"
```

vector flag

This flag tells Perl to interpret the supplied string as a vector of integers, one for each character in the string. Perl applies the format to each integer in turn, then joins the resulting strings with a separator (a dot `.` by default). This can be useful for displaying ordinal values of characters in arbitrary strings:

```
printf "%vd", "AB\x{100}";           # prints "65.66.256"
printf "version is v%vd\n", $^V;     # Perl's version
```

Put an asterisk `*` before the `v` to override the string to use to separate the numbers:

```
printf "address is %*vX\n", ":", $addr;   # IPv6 address
printf "bits are %0*v8b\n", " ", $bits;   # random bitstring
```

You can also explicitly specify the argument number to use for the join string using something like `*2$v`; for example:

```
printf '%*4$vX %*4$vX %*4$vX',       # 3 IPv6 addresses
        @addr[1..3], ":";
```

(minimum) width

Arguments are usually formatted to be only as wide as required to display the given value. You can override the width by putting a number here, or get the width from the next argument (with `*`) or from a specified argument (e.g., with `*2$`):

```
printf "<%s>", "a";       # prints "<a>"
printf "<%6s>", "a";      # prints "<     a>"
printf "<%*s>", 6, "a";   # prints "<     a>"
printf '<%*2$s>', "a", 6; # prints "<     a>"
printf "<%2s>", "long";   # prints "<long>" (does not truncate)
```

If a field width obtained through `*` is negative, it has the same effect as the `-` flag: left-justification.

precision, or maximum width

You can specify a precision (for numeric conversions) or a maximum width (for string conversions) by specifying a `.` followed by a number. For floating-point formats except `g` and `G`, this specifies how many places right of the decimal point to show (the default being 6). For example:

```
# these examples are subject to system-specific variation
printf '<%f>', 1;    # prints "<1.000000>"
printf '<%.1f>', 1;  # prints "<1.0>"
printf '<%.0f>', 1;  # prints "<1>"
printf '<%e>', 10;   # prints "<1.000000e+01>"
printf '<%.1e>', 10; # prints "<1.0e+01>"
```

For “g” and “G”, this specifies the maximum number of significant digits to show; for example:

```
# These examples are subject to system-specific variation.
printf '<%g>', 1;        # prints "<1>"
printf '<%.10g>', 1;     # prints "<1>"
printf '<%g>', 100;      # prints "<100>"
printf '<%.1g>', 100;    # prints "<1e+02>"
printf '<%.2g>', 100.01; # prints "<1e+02>"
printf '<%.5g>', 100.01; # prints "<100.01>"
printf '<%.4g>', 100.01; # prints "<100>"
printf '<%.1g>', 0.0111; # prints "<0.01>"
printf '<%.2g>', 0.0111; # prints "<0.011>"
printf '<%.3g>', 0.0111; # prints "<0.0111>"
```

For integer conversions, specifying a precision implies that the output of the number itself should be zero-padded to this width, where the 0 flag is ignored:

```
printf '<%.6d>', 1;      # prints "<000001>"
printf '<%+.6d>', 1;     # prints "<+000001>"
printf '<%-10.6d>', 1;   # prints "<000001    >"
printf '<%10.6d>', 1;    # prints "<    000001>"
printf '<%010.6d>', 1;   # prints "<    000001>"
printf '<%+10.6d>', 1;   # prints "<   +000001>"

printf '<%.6x>', 1;      # prints "<000001>"
printf '<%#.6x>', 1;     # prints "<0x000001>"
printf '<%-10.6x>', 1;   # prints "<000001    >"
printf '<%10.6x>', 1;    # prints "<    000001>"
printf '<%010.6x>', 1;   # prints "<    000001>"
printf '<%#10.6x>', 1;   # prints "<  0x000001>"
```

For string conversions, specifying a precision truncates the string to fit the specified width:

```
printf '<%.5s>', "truncated";   # prints "<trunc>"
printf '<%10.5s>', "truncated"; # prints "<     trunc>"
```

You can also get the precision from the next argument using `.*`, or from a specified argument (e.g., with `.*2$`):

```
printf '<%.6x>', 1;       # prints "<000001>"
printf '<%.*x>', 6, 1;    # prints "<000001>"

printf '<%.*2$x>', 1, 6;  # prints "<000001>"

printf '<%6.*2$x>', 1, 4; # prints "<  0001>"
```

If a precision obtained through `*` is negative, it counts as having no precision at all.

```
printf '<%.*s>',  7, "string";   # prints "<string>"
printf '<%.*s>',  3, "string";   # prints "<str>"
printf '<%.*s>',  0, "string";   # prints "<>"
printf '<%.*s>', -1, "string";   # prints "<string>"

printf '<%.*d>',  1, 0;   # prints "<0>"
printf '<%.*d>',  0, 0;   # prints "<>"
printf '<%.*d>', -1, 0;   # prints "<0>"
```

size

For numeric conversions, you can specify the size to interpret the number as using `l`, `h`, `V`, `q`, `L`, or `ll`. For integer conversions (`d u o x X b i D U O`), numbers are usually assumed to be whatever the default integer size is on your platform (usually 32 or 64 bits), but you can override this to use instead one of the standard C types, as supported by the compiler used to build Perl:

```
hh           interpret integer as C type "char" or "unsigned
             char" on Perl 5.14 or later
h            interpret integer as C type "short" or
             "unsigned short"
j            interpret integer as C type "intmax_t" on Perl
             5.14 or later, and only with a C99 compiler
             (unportable)
l            interpret integer as C type "long" or
             "unsigned long"
q, L, or ll interpret integer as C type "long long",
             "unsigned long long", or "quad" (typically
             64-bit integers)
t            interpret integer as C type "ptrdiff_t" on Perl
             5.14 or later
z            interpret integer as C type "size_t" on Perl 5.14
             or later
```

As of 5.14, none of these raises an exception if they are not supported on your platform. However, if warnings are enabled, a warning of the `printf` warning class is issued on an unsupported conversion flag. Should you instead prefer an exception, do this:

```
use warnings FATAL => "printf";
```

If you would like to know about a version dependency before you start running the program, put something like this at its top:

```
use 5.014;  # for hh/j/t/z/ printf modifiers
```

You can find out whether your Perl supports quads via Config:

```
use Config;
if ($Config{use64bitint} eq "define"
    || $Config{longsize} >= 8) {
    print "Nice quads!\n";
}
```

For floating-point conversions (`e f g E F G`), numbers are usually assumed to be the default floating-point size on your platform (double or long double), but you can force "long double" with `q`, `L`, or `ll` if your platform supports them. You can find out whether your Perl supports long doubles via Config:

```
use Config;
print "long doubles\n" if $Config{d_longdbl} eq "define";
```

You can find out whether Perl considers "long double" to be the default floating-point size to use on your platform via Config:

```
use Config;
if ($Config{uselongdouble} eq "define") {
    print "long doubles by default\n";
}
```

It can also be that long doubles and doubles are the same thing:

```
use Config;
($Config{doublesize} == $Config{longdblsize}) &&
        print "doubles are long doubles\n";
```

The size specifier `V` has no effect for Perl code, but is supported for compatibility with XS code. It means "use the standard size for a Perl integer or floating-point number", which is the default.

order of arguments

Normally, `sprintf` takes the next unused argument as the value to format for each format specification. If the format specification uses `*` to require additional arguments, these are consumed from the argument list in the order they appear in the format specification *before* the value to format. Where an argument is specified by an explicit index, this does not affect the normal order for the arguments, even when the explicitly specified index would have been the next argument.

So:

```
printf "<%*.*s>", $a, $b, $c;
```

uses `$a` for the width, `$b` for the precision, and `$c` as the value to format; while:

```
printf '<%*1$.*s>', $a, $b;
```

would use `$a` for the width and precision, and `$b` as the value to format.

Here are some more examples; be aware that when using an explicit index, the `$` may need escaping:

```
printf "%2\$d %d\n",      12, 34;     # will print "34 12\n"
printf "%2\$d %d %d\n",   12, 34;     # will print "34 12 34\n"
printf "%3\$d %d %d\n",   12, 34, 56; # will print "56 12 34\n"
printf "%2\$*3\$d %d\n",  12, 34,  3; # will print " 34 12\n"
printf "%*1\$.*f\n",       4,  5, 10; # will print "5.0000\n"
```

If `use locale` (including `use locale ':not_characters'`) is in effect and `POSIX::setlocale` has been called, the character used for the decimal separator in formatted floating-point numbers is affected by the `LC_NUMERIC` locale. See perllocale and POSIX.

sqrt EXPR

sqrt Return the positive square root of EXPR. If EXPR is omitted, uses `$_`. Works only for non-negative operands unless you've loaded the `Math::Complex` module.

```
use Math::Complex;
print sqrt(-4);    # prints 2i
```

srand EXPR

srand

Sets and returns the random number seed for the `rand` operator.

The point of the function is to "seed" the `rand` function so that `rand` can produce a different sequence each time you run your program. When called with a parameter, `srand` uses that for the seed; otherwise it (semi-)randomly chooses a seed. In either case, starting with Perl 5.14, it returns the seed. To signal that your code will work *only* on Perls of a recent vintage:

```
use 5.014;  # so srand returns the seed
```

If `srand` is not called explicitly, it is called implicitly without a parameter at the first use of the `rand` operator. However, there are a few situations where programs are likely to want to call `srand`. One is for generating predictable results, generally for testing or debugging. There, you use `srand($seed)`, with the same `$seed` each time. Another case is that you may want to call `srand` after a `fork` to avoid child processes sharing the same seed value as the parent (and consequently each other).

Do **not** call `srand()` (i.e., without an argument) more than once per process. The internal state of the random number generator should contain more entropy than can be provided by any seed, so calling `srand` again actually *loses* randomness.

Most implementations of `srand` take an integer and will silently truncate decimal numbers. This means `srand(42)` will usually produce the same results as `srand(42.1)`. To be safe, always pass `srand` an integer.

A typical use of the returned seed is for a test program which has too many combinations to test comprehensively in the time available to it each run. It can test a random subset each time, and should there be a failure, log the seed used for that run so that it can later be used to reproduce the same

results.

`rand` is not cryptographically secure. You should not rely on it in security-sensitive situations. As of this writing, a number of third-party CPAN modules offer random number generators intended by their authors to be cryptographically secure, including: Data::Entropy, Crypt::Random, Math::Random::Secure, and Math::TrulyRandom.

stat FILEHANDLE
stat EXPR
stat DIRHANDLE
stat Returns a 13-element list giving the status info for a file, either the file opened via FILEHANDLE or DIRHANDLE, or named by EXPR. If EXPR is omitted, it stats `$_` (not _!). Returns the empty list if `stat` fails. Typically used as follows:

```
my ($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
    $atime,$mtime,$ctime,$blksize,$blocks)
       = stat($filename);
```

Not all fields are supported on all filesystem types. Here are the meanings of the fields:

```
 0 dev      device number of filesystem
 1 ino      inode number
 2 mode     file mode  (type and permissions)
 3 nlink    number of (hard) links to the file
 4 uid      numeric user ID of file's owner
 5 gid      numeric group ID of file's owner
 6 rdev     the device identifier (special files only)
 7 size     total size of file, in bytes
 8 atime    last access time in seconds since the epoch
 9 mtime    last modify time in seconds since the epoch
10 ctime    inode change time in seconds since the epoch (*)
11 blksize  preferred I/O size in bytes for interacting with the
            file (may vary from file to file)
12 blocks   actual number of system-specific blocks allocated
            on disk (often, but not always, 512 bytes each)
```

(The epoch was at 00:00 January 1, 1970 GMT.)

(*) Not all fields are supported on all filesystem types. Notably, the ctime field is non-portable. In particular, you cannot expect it to be a ''creation time''; see ''Files and Filesystems'' in perlport for details.

If `stat` is passed the special filehandle consisting of an underline, no stat is done, but the current contents of the stat structure from the last `stat`, `lstat`, or filetest are returned. Example:

```
if (-x $file && (($d) = stat(_)) && $d < 0) {
    print "$file is executable NFS file\n";
}
```

(This works on machines only for which the device number is negative under NFS.)

Because the mode contains both the file type and its permissions, you should mask off the file type portion and (s)printf using a `"%o"` if you want to see the real permissions.

```
my $mode = (stat($filename))[2];
printf "Permissions are %04o\n", $mode & 07777;
```

In scalar context, `stat` returns a boolean value indicating success or failure, and, if successful, sets the information associated with the special filehandle _.

The File::stat module provides a convenient, by-name access mechanism:

```
use File::stat;
my $sb = stat($filename);
printf "File is %s, size is %s, perm %04o, mtime %s\n",
       $filename, $sb->size, $sb->mode & 07777,
       scalar localtime $sb->mtime;
```

You can import symbolic mode constants (`S_IF*`) and functions (`S_IS*`) from the Fcntl module:

```
use Fcntl ':mode';

my $mode = (stat($filename))[2];

my $user_rwx      = ($mode & S_IRWXU) >> 6;
my $group_read    = ($mode & S_IRGRP) >> 3;
my $other_execute =  $mode & S_IXOTH;

printf "Permissions are %04o\n", S_IMODE($mode), "\n";

my $is_setuid     =  $mode & S_ISUID;
my $is_directory  =  S_ISDIR($mode);
```

You could write the last two using the `-u` and `-d` operators. Commonly available `S_IF*` constants are:

```
# Permissions: read, write, execute, for user, group, others.

S_IRWXU S_IRUSR S_IWUSR S_IXUSR
S_IRWXG S_IRGRP S_IWGRP S_IXGRP
S_IRWXO S_IROTH S_IWOTH S_IXOTH

# Setuid/Setgid/Stickiness/SaveText.
# Note that the exact meaning of these is system-dependent.

S_ISUID S_ISGID S_ISVTX S_ISTXT

# File types.  Not all are necessarily available on
# your system.

S_IFREG S_IFDIR S_IFLNK S_IFBLK S_IFCHR
S_IFIFO S_IFSOCK S_IFWHT S_ENFMT

# The following are compatibility aliases for S_IRUSR,
# S_IWUSR, and S_IXUSR.

S_IREAD S_IWRITE S_IEXEC
```

and the `S_IF*` functions are

```
S_IMODE($mode)    the part of $mode containing the permission
                  bits and the setuid/setgid/sticky bits

S_IFMT($mode)     the part of $mode containing the file type
                  which can be bit-anded with (for example)
                  S_IFREG or with the following functions

# The operators -f, -d, -l, -b, -c, -p, and -S.

S_ISREG($mode) S_ISDIR($mode) S_ISLNK($mode)
S_ISBLK($mode) S_ISCHR($mode) S_ISFIFO($mode) S_ISSOCK($mode)

# No direct -X operator counterpart, but for the first one
# the -g operator is often equivalent.  The ENFMT stands for
```

```
    # record flocking enforcement, a platform-dependent feature.

    S_ISENFMT($mode) S_ISWHT($mode)
```

See your native *chmod*(2) and *stat*(2) documentation for more details about the `S_*` constants. To get status info for a symbolic link instead of the target file behind the link, use the `lstat` function.

Portability issues: "stat" in perlport.

state VARLIST
state TYPE VARLIST
state VARLIST : ATTRS
state TYPE VARLIST : ATTRS

`state` declares a lexically scoped variable, just like `my`. However, those variables will never be reinitialized, contrary to lexical variables that are reinitialized each time their enclosing block is entered. See "Persistent Private Variables" in perlsub for details.

If more than one variable is listed, the list must be placed in parentheses. With a parenthesised list, `undef` can be used as a dummy placeholder. However, since initialization of state variables in list context is currently not possible this would serve no purpose.

`state` is available only if the `"state"` feature is enabled or if it is prefixed with `CORE::`. The `"state"` feature is enabled automatically with a `use v5.10` (or higher) declaration in the current scope.

study SCALAR
study

At this time, `study` does nothing. This may change in the future.

Prior to Perl version 5.16, it would create an inverted index of all characters that occurred in the given SCALAR (or `$_` if unspecified). When matching a pattern, the rarest character from the pattern would be looked up in this index. Rarity was based on some static frequency tables constructed from some C programs and English text.

sub NAME BLOCK
sub NAME (PROTO) BLOCK
sub NAME : ATTRS BLOCK
sub NAME (PROTO) : ATTRS BLOCK

This is subroutine definition, not a real function *per se*. Without a BLOCK it's just a forward declaration. Without a NAME, it's an anonymous function declaration, so does return a value: the CODE ref of the closure just created.

See perlsub and perlref for details about subroutines and references; see attributes and Attribute::Handlers for more information about attributes.

__SUB__

A special token that returns a reference to the current subroutine, or `undef` outside of a subroutine.

The behaviour of `__SUB__` within a regex code block (such as `/(?{...})/`) is subject to change.

This token is only available under `use v5.16` or the `"current_sub"` feature. See feature.

substr EXPR,OFFSET,LENGTH,REPLACEMENT
substr EXPR,OFFSET,LENGTH
substr EXPR,OFFSET

Extracts a substring out of EXPR and returns it. First character is at offset zero. If OFFSET is negative, starts that far back from the end of the string. If LENGTH is omitted, returns everything through the end of the string. If LENGTH is negative, leaves that many characters off the end of the string.

```
    my $s = "The black cat climbed the green tree";
    my $color  = substr $s, 4, 5;      # black
    my $middle = substr $s, 4, -11;    # black cat climbed the
    my $end    = substr $s, 14;        # climbed the green tree
    my $tail   = substr $s, -4;        # tree
    my $z      = substr $s, -4, 2;     # tr
```

You can use the `substr` function as an lvalue, in which case EXPR must itself be an lvalue. If you

assign something shorter than LENGTH, the string will shrink, and if you assign something longer than LENGTH, the string will grow to accommodate it. To keep the string the same length, you may need to pad or chop your value using `sprintf`.

If OFFSET and LENGTH specify a substring that is partly outside the string, only the part within the string is returned. If the substring is beyond either end of the string, `substr` returns the undefined value and produces a warning. When used as an lvalue, specifying a substring that is entirely outside the string raises an exception. Here's an example showing the behavior for boundary cases:

```
my $name = 'fred';
substr($name, 4) = 'dy';         # $name is now 'freddy'
my $null = substr $name, 6, 2;   # returns "" (no warning)
my $oops = substr $name, 7;      # returns undef, with warning
substr($name, 7) = 'gap';        # raises an exception
```

An alternative to using `substr` as an lvalue is to specify the replacement string as the 4th argument. This allows you to replace parts of the EXPR and return what was there before in one operation, just as you can with `splice`.

```
my $s = "The black cat climbed the green tree";
my $z = substr $s, 14, 7, "jumped from";    # climbed
# $s is now "The black cat jumped from the green tree"
```

Note that the lvalue returned by the three-argument version of `substr` acts as a 'magic bullet'; each time it is assigned to, it remembers which part of the original string is being modified; for example:

```
my $x = '1234';
for (substr($x,1,2)) {
    $_ = 'a';   print $x,"\n";    # prints 1a4
    $_ = 'xyz'; print $x,"\n";    # prints 1xyz4
    $x = '56789';
    $_ = 'pq';  print $x,"\n";    # prints 5pq9
}
```

With negative offsets, it remembers its position from the end of the string when the target string is modified:

```
my $x = '1234';
for (substr($x, -3, 2)) {
    $_ = 'a';   print $x,"\n";    # prints 1a4, as above
    $x = 'abcdefg';
    print $_,"\n";                # prints f
}
```

Prior to Perl version 5.10, the result of using an lvalue multiple times was unspecified. Prior to 5.16, the result with negative offsets was unspecified.

symlink OLDFILE,NEWFILE

Creates a new filename symbolically linked to the old filename. Returns `1` for success, `0` otherwise. On systems that don't support symbolic links, raises an exception. To check for that, use eval:

```
my $symlink_exists = eval { symlink("",""); 1 };
```

Portability issues: "symlink" in perlport.

syscall NUMBER, LIST

Calls the system call specified as the first element of the list, passing the remaining elements as arguments to the system call. If unimplemented, raises an exception. The arguments are interpreted as follows: if a given argument is numeric, the argument is passed as an int. If not, the pointer to the string value is passed. You are responsible to make sure a string is pre-extended long enough to receive any result that might be written into a string. You can't use a string literal (or other read-only string) as an argument to `syscall` because Perl has to assume that any string pointer might be written through. If your integer arguments are not literals and have never been interpreted in a numeric context, you may need to add `0` to them to force them to look like numbers. This emulates the `syswrite` function (or vice versa):

```
require 'syscall.ph';        # may need to run h2ph
my $s = "hi there\n";
syscall(SYS_write(), fileno(STDOUT), $s, length $s);
```

Note that Perl supports passing of up to only 14 arguments to your syscall, which in practice should (usually) suffice.

Syscall returns whatever value returned by the system call it calls. If the system call fails, `syscall` returns `-1` and sets `$!` (errno). Note that some system calls *can* legitimately return `-1`. The proper way to handle such calls is to assign `$! = 0` before the call, then check the value of `$!` if `syscall` returns `-1`.

There's a problem with `syscall(SYS_pipe())`: it returns the file number of the read end of the pipe it creates, but there is no way to retrieve the file number of the other end. You can avoid this problem by using `pipe` instead.

Portability issues: "syscall" in perlport.

sysopen FILEHANDLE,FILENAME,MODE
sysopen FILEHANDLE,FILENAME,MODE,PERMS

Opens the file whose filename is given by FILENAME, and associates it with FILEHANDLE. If FILEHANDLE is an expression, its value is used as the real filehandle wanted; an undefined scalar will be suitably autovivified. This function calls the underlying operating system's *open* (2) function with the parameters FILENAME, MODE, and PERMS.

Returns true on success and `undef` otherwise.

The possible values and flag bits of the MODE parameter are system-dependent; they are available via the standard module `Fcntl`. See the documentation of your operating system's *open* (2) syscall to see which values and flag bits are available. You may combine several flags using the `|`-operator.

Some of the most common values are `O_RDONLY` for opening the file in read-only mode, `O_WRONLY` for opening the file in write-only mode, and `O_RDWR` for opening the file in read-write mode.

For historical reasons, some values work on almost every system supported by Perl: 0 means read-only, 1 means write-only, and 2 means read/write. We know that these values do *not* work under OS/390 and on the Macintosh; you probably don't want to use them in new code.

If the file named by FILENAME does not exist and the `open` call creates it (typically because MODE includes the `O_CREAT` flag), then the value of PERMS specifies the permissions of the newly created file. If you omit the PERMS argument to `sysopen`, Perl uses the octal value `0666`. These permission values need to be in octal, and are modified by your process's current `umask`.

In many systems the `O_EXCL` flag is available for opening files in exclusive mode. This is **not** locking: exclusiveness means here that if the file already exists, `sysopen` fails. `O_EXCL` may not work on network filesystems, and has no effect unless the `O_CREAT` flag is set as well. Setting `O_CREAT|O_EXCL` prevents the file from being opened if it is a symbolic link. It does not protect against symbolic links in the file's path.

Sometimes you may want to truncate an already-existing file. This can be done using the `O_TRUNC` flag. The behavior of `O_TRUNC` with `O_RDONLY` is undefined.

You should seldom if ever use `0644` as argument to `sysopen`, because that takes away the user's option to have a more permissive umask. Better to omit it. See `umask` for more on this.

Note that under Perls older than 5.8.0, `sysopen` depends on the *fdopen* (3) C library function. On many Unix systems, *fdopen* (3) is known to fail when file descriptors exceed a certain value, typically 255. If you need more file descriptors than that, consider using the `POSIX::open` function. For Perls 5.8.0 and later, PerlIO is (most often) the default.

See perlopentut for a kinder, gentler explanation of opening files.

Portability issues: "sysopen" in perlport.

sysread FILEHANDLE,SCALAR,LENGTH,OFFSET

sysread FILEHANDLE,SCALAR,LENGTH

Attempts to read LENGTH bytes of data into variable SCALAR from the specified FILEHANDLE, using *read*(2). It bypasses buffered IO, so mixing this with other kinds of reads, `print`, `write`, `seek`, `tell`, or `eof` can cause confusion because the perlio or stdio layers usually buffer data. Returns the number of bytes actually read, `0` at end of file, or undef if there was an error (in the latter case `$!` is also set). SCALAR will be grown or shrunk so that the last byte actually read is the last byte of the scalar after the read.

An OFFSET may be specified to place the read data at some place in the string other than the beginning. A negative OFFSET specifies placement at that many characters counting backwards from the end of the string. A positive OFFSET greater than the length of SCALAR results in the string being padded to the required size with `"\0"` bytes before the result of the read is appended.

There is no *syseof()* function, which is ok, since `eof` doesn't work well on device files (like ttys) anyway. Use `sysread` and check for a return value for 0 to decide whether you're done.

Note that if the filehandle has been marked as `:utf8`, Unicode characters are read instead of bytes (the LENGTH, OFFSET, and the return value of `sysread` are in Unicode characters). The `:encoding(...)` layer implicitly introduces the `:utf8` layer. See `binmode`, `open`, and the open pragma.

sysseek FILEHANDLE,POSITION,WHENCE

Sets FILEHANDLE's system position *in bytes* using *lseek*(2). FILEHANDLE may be an expression whose value gives the name of the filehandle. The values for WHENCE are `0` to set the new position to POSITION; `1` to set the it to the current position plus POSITION; and `2` to set it to EOF plus POSITION, typically negative.

Note the emphasis on bytes: even if the filehandle has been set to operate on characters (for example using the `:encoding(UTF-8)` I/O layer), the `seek`, `tell`, and `sysseek` family of functions use byte offsets, not character offsets, because seeking to a character offset would be very slow in a UTF-8 file.

`sysseek` bypasses normal buffered IO, so mixing it with reads other than `sysread` (for example `readline` or `read`), `print`, `write`, `seek`, `tell`, or `eof` may cause confusion.

For WHENCE, you may also use the constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` (start of the file, current position, end of the file) from the Fcntl module. Use of the constants is also more portable than relying on 0, 1, and 2. For example to define a "systell" function:

```
use Fcntl 'SEEK_CUR';
sub systell { sysseek($_[0], 0, SEEK_CUR) }
```

Returns the new position, or the undefined value on failure. A position of zero is returned as the string `"0 but true"`; thus `sysseek` returns true on success and false on failure, yet you can still easily determine the new position.

system LIST

system PROGRAM LIST

Does exactly the same thing as `exec`, except that a fork is done first and the parent process waits for the child process to exit. Note that argument processing varies depending on the number of arguments. If there is more than one argument in LIST, or if LIST is an array with more than one value, starts the program given by the first element of the list with arguments given by the rest of the list. If there is only one scalar argument, the argument is checked for shell metacharacters, and if there are any, the entire argument is passed to the system's command shell for parsing (this is `/bin/sh -c` on Unix platforms, but varies on other platforms). If there are no shell metacharacters in the argument, it is split into words and passed directly to `execvp`, which is more efficient. On Windows, only the `system PROGRAM LIST` syntax will reliably avoid using the shell; `system LIST`, even with more than one element, will fall back to the shell if the first spawn fails.

Perl will attempt to flush all files opened for output before any operation that may do a fork, but this may not be supported on some platforms (see perlport). To be safe, you may need to set `$|` (`$AUTOFLUSH` in English) or call the `autoflush` method of `IO::Handle` on any open handles.

The return value is the exit status of the program as returned by the `wait` call. To get the actual exit value, shift right by eight (see below). See also `exec`. This is *not* what you want to use to capture the

output from a command; for that you should use merely backticks or `qx//`, as described in "'STRING'" in perlop. Return value of -1 indicates a failure to start the program or an error of the *wait*(2) system call (inspect `$!` for the reason).

If you'd like to make `system` (and many other bits of Perl) die on error, have a look at the autodie pragma.

Like `exec`, `system` allows you to lie to a program about its name if you use the `system PROGRAM LIST` syntax. Again, see `exec`.

Since `SIGINT` and `SIGQUIT` are ignored during the execution of `system`, if you expect your program to terminate on receipt of these signals you will need to arrange to do so yourself based on the return value.

```
my @args = ("command", "arg1", "arg2");
system(@args) == 0
    or die "system @args failed: $?";
```

If you'd like to manually inspect `system`'s failure, you can check all possible failure modes by inspecting `$?` like this:

```
if ($? == -1) {
    print "failed to execute: $!\n";
}
elsif ($? & 127) {
    printf "child died with signal %d, %s coredump\n",
        ($? & 127),  ($? & 128) ? 'with' : 'without';
}
else {
    printf "child exited with value %d\n", $? >> 8;
}
```

Alternatively, you may inspect the value of `${^CHILD_ERROR_NATIVE}` with the `W*()` calls from the POSIX module.

When `system`'s arguments are executed indirectly by the shell, results and return codes are subject to its quirks. See "'STRING'" in perlop and `exec` for details.

Since `system` does a `fork` and `wait` it may affect a `SIGCHLD` handler. See perlipc for details.

Portability issues: "system" in perlport.

syswrite FILEHANDLE,SCALAR,LENGTH,OFFSET
syswrite FILEHANDLE,SCALAR,LENGTH
syswrite FILEHANDLE,SCALAR

Attempts to write LENGTH bytes of data from variable SCALAR to the specified FILEHANDLE, using *write*(2). If LENGTH is not specified, writes whole SCALAR. It bypasses buffered IO, so mixing this with reads (other than `sysread()`), `print`, `write`, `seek`, `tell`, or `eof` may cause confusion because the perlio and stdio layers usually buffer data. Returns the number of bytes actually written, or `undef` if there was an error (in this case the errno variable `$!` is also set). If the LENGTH is greater than the data available in the SCALAR after the OFFSET, only as much data as is available will be written.

An OFFSET may be specified to write the data from some part of the string other than the beginning. A negative OFFSET specifies writing that many characters counting backwards from the end of the string. If SCALAR is of length zero, you can only use an OFFSET of 0.

WARNING: If the filehandle is marked `:utf8`, Unicode characters encoded in UTF-8 are written instead of bytes, and the LENGTH, OFFSET, and return value of `syswrite` are in (UTF8-encoded Unicode) characters. The `:encoding(...)` layer implicitly introduces the `:utf8` layer. Alternately, if the handle is not marked with an encoding but you attempt to write characters with code points over 255, raises an exception. See `binmode`, `open`, and the open pragma.

tell FILEHANDLE
tell

Returns the current position *in bytes* for FILEHANDLE, or -1 on error. FILEHANDLE may be an expression whose value gives the name of the actual filehandle. If FILEHANDLE is omitted, assumes

the file last read.

Note the emphasis on bytes: even if the filehandle has been set to operate on characters (for example using the `:encoding(UTF-8)` I/O layer), the `seek`, `tell`, and `sysseek` family of functions use byte offsets, not character offsets, because seeking to a character offset would be very slow in a UTF–8 file.

The return value of `tell` for the standard streams like the STDIN depends on the operating system: it may return –1 or something else. `tell` on pipes, fifos, and sockets usually returns –1.

There is no `systell` function. Use `sysseek($fh, 0, 1)` for that.

Do not use `tell` (or other buffered I/O operations) on a filehandle that has been manipulated by `sysread`, `syswrite`, or `sysseek`. Those functions ignore the buffering, while `tell` does not.

telldir DIRHANDLE

Returns the current position of the `readdir` routines on DIRHANDLE. Value may be given to `seekdir` to access a particular location in a directory. `telldir` has the same caveats about possible directory compaction as the corresponding system library routine.

tie VARIABLE,CLASSNAME,LIST

This function binds a variable to a package class that will provide the implementation for the variable. VARIABLE is the name of the variable to be enchanted. CLASSNAME is the name of a class implementing objects of correct type. Any additional arguments are passed to the appropriate constructor method of the class (meaning `TIESCALAR`, `TIEHANDLE`, `TIEARRAY`, or `TIEHASH`). Typically these are arguments such as might be passed to the *dbm_open*(3) function of C. The object returned by the constructor is also returned by the `tie` function, which would be useful if you want to access other methods in CLASSNAME.

Note that functions such as `keys` and `values` may return huge lists when used on large objects, like DBM files. You may prefer to use the `each` function to iterate over such. Example:

```
# print out history file offsets
use NDBM_File;
tie(my %HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (my ($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L', $val), "\n";
}
```

A class implementing a hash should have the following methods:

```
TIEHASH classname, LIST
FETCH this, key
STORE this, key, value
DELETE this, key
CLEAR this
EXISTS this, key
FIRSTKEY this
NEXTKEY this, lastkey
SCALAR this
DESTROY this
UNTIE this
```

A class implementing an ordinary array should have the following methods:

```
TIEARRAY classname, LIST
FETCH this, key
STORE this, key, value
FETCHSIZE this
STORESIZE this, count
CLEAR this
PUSH this, LIST
POP this
SHIFT this
UNSHIFT this, LIST
SPLICE this, offset, length, LIST
EXTEND this, count
DELETE this, key
EXISTS this, key
DESTROY this
UNTIE this
```

A class implementing a filehandle should have the following methods:

```
TIEHANDLE classname, LIST
READ this, scalar, length, offset
READLINE this
GETC this
WRITE this, scalar, length, offset
PRINT this, LIST
PRINTF this, format, LIST
BINMODE this
EOF this
FILENO this
SEEK this, position, whence
TELL this
OPEN this, mode, LIST
CLOSE this
DESTROY this
UNTIE this
```

A class implementing a scalar should have the following methods:

```
TIESCALAR classname, LIST
FETCH this,
STORE this, value
DESTROY this
UNTIE this
```

Not all methods indicated above need be implemented. See perltie, Tie::Hash, Tie::Array, Tie::Scalar, and Tie::Handle.

Unlike `dbmopen`, the `tie` function will not `use` or `require` a module for you; you need to do that explicitly yourself. See DB_File or the Config module for interesting `tie` implementations.

For further details see perltie, `tied`.

tied VARIABLE
: Returns a reference to the object underlying VARIABLE (the same value that was originally returned by the `tie` call that bound the variable to a package.) Returns the undefined value if VARIABLE isn't tied to a package.

time
: Returns the number of non-leap seconds since whatever time the system considers to be the epoch, suitable for feeding to `gmtime` and `localtime`. On most systems the epoch is 00:00:00 UTC, January 1, 1970; a prominent exception being Mac OS Classic which uses 00:00:00, January 1, 1904 in the current local time zone for its epoch.

  For measuring time in better granularity than one second, use the Time::HiRes module from Perl 5.8

onwards (or from CPAN before then), or, if you have *gettimeofday*(2), you may be able to use the `syscall` interface of Perl. See perlfaq8 for details.

For date and time processing look at the many related modules on CPAN. For a comprehensive date and time representation look at the DateTime module.

times
Returns a four-element list giving the user and system times in seconds for this process and any exited children of this process.

```
my ($user,$system,$cuser,$csystem) = times;
```

In scalar context, `times` returns `$user`.

Children's times are only included for terminated children.

Portability issues: "times" in perlport.

tr/// The transliteration operator. Same as `y///`. See "Quote-Like Operators" in perlop.

truncate FILEHANDLE,LENGTH
truncate EXPR,LENGTH
Truncates the file opened on FILEHANDLE, or named by EXPR, to the specified length. Raises an exception if truncate isn't implemented on your system. Returns true if successful, `undef` on error.

The behavior is undefined if LENGTH is greater than the length of the file.

The position in the file of FILEHANDLE is left unchanged. You may want to call seek before writing to the file.

Portability issues: "truncate" in perlport.

uc EXPR
uc Returns an uppercased version of EXPR. This is the internal function implementing the `\U` escape in double-quoted strings. It does not attempt to do titlecase mapping on initial letters. See `ucfirst` for that.

If EXPR is omitted, uses `$_`.

This function behaves the same way under various pragmas, such as in a locale, as `lc` does.

ucfirst EXPR
ucfirst
Returns the value of EXPR with the first character in uppercase (titlecase in Unicode). This is the internal function implementing the `\u` escape in double-quoted strings.

If EXPR is omitted, uses `$_`.

This function behaves the same way under various pragmas, such as in a locale, as `lc` does.

umask EXPR
umask
Sets the umask for the process to EXPR and returns the previous value. If EXPR is omitted, merely returns the current umask.

The Unix permission `rwxr-x---` is represented as three sets of three bits, or three octal digits: `0750` (the leading 0 indicates octal and isn't one of the digits). The `umask` value is such a number representing disabled permissions bits. The permission (or "mode") values you pass `mkdir` or `sysopen` are modified by your umask, so even if you tell `sysopen` to create a file with permissions `0777`, if your umask is `0022`, then the file will actually be created with permissions `0755`. If your `umask` were `0027` (group can't write; others can't read, write, or execute), then passing `sysopen` `0666` would create a file with mode `0640` (because `0666 &~ 027` is `0640`).

Here's some advice: supply a creation mode of `0666` for regular files (in `sysopen`) and one of `0777` for directories (in `mkdir`) and executable files. This gives users the freedom of choice: if they want protected files, they might choose process umasks of `022`, `027`, or even the particularly antisocial mask of `077`. Programs should rarely if ever make policy decisions better left to the user. The exception to this is when writing files that should be kept private: mail files, web browser cookies, *.rhosts* files, and so on.

If *umask*(2) is not implemented on your system and you are trying to restrict access for *yourself* (i.e., `(EXPR & 0700) > 0`), raises an exception. If *umask*(2) is not implemented and you are not trying to restrict access for yourself, returns `undef`.

Remember that a umask is a number, usually given in octal; it is *not* a string of octal digits. See also `oct`, if all you have is a string.

Portability issues: "umask" in perlport.

undef EXPR
undef

Undefines the value of EXPR, which must be an lvalue. Use only on a scalar value, an array (using `@`), a hash (using `%`), a subroutine (using `&`), or a typeglob (using `*`). Saying `undef $hash{$key}` will probably not do what you expect on most predefined variables or DBM list values, so don't do that; see `delete`. Always returns the undefined value. You can omit the EXPR, in which case nothing is undefined, but you still get an undefined value that you could, for instance, return from a subroutine, assign to a variable, or pass as a parameter. Examples:

```
undef $foo;
undef $bar{'blurfl'};      # Compare to: delete $bar{'blurfl'};
undef @ary;
undef %hash;
undef &mysub;
undef *xyz;       # destroys $xyz, @xyz, %xyz, &xyz, etc.
return (wantarray ? (undef, $errmsg) : undef) if $they_blew_it;
select undef, undef, undef, 0.25;
my ($x, $y, undef, $z) = foo();    # Ignore third value returned
```

Note that this is a unary operator, not a list operator.

unlink LIST
unlink

Deletes a list of files. On success, it returns the number of files it successfully deleted. On failure, it returns false and sets `$!` (errno):

```
my $unlinked = unlink 'a', 'b', 'c';
unlink @goners;
unlink glob "*.bak";
```

On error, `unlink` will not tell you which files it could not remove. If you want to know which files you could not remove, try them one at a time:

```
foreach my $file ( @goners ) {
    unlink $file or warn "Could not unlink $file: $!";
}
```

Note: `unlink` will not attempt to delete directories unless you are superuser and the **-U** flag is supplied to Perl. Even if these conditions are met, be warned that unlinking a directory can inflict damage on your filesystem. Finally, using `unlink` on directories is not supported on many operating systems. Use `rmdir` instead.

If LIST is omitted, `unlink` uses `$_`.

unpack TEMPLATE,EXPR
unpack TEMPLATE

`unpack` does the reverse of `pack`: it takes a string and expands it out into a list of values. (In scalar context, it returns merely the first value produced.)

If EXPR is omitted, unpacks the `$_` string. See perlpacktut for an introduction to this function.

The string is broken into chunks described by the TEMPLATE. Each chunk is converted separately to a value. Typically, either the string is a result of `pack`, or the characters of the string represent a C structure of some kind.

The TEMPLATE has the same format as in the `pack` function. Here's a subroutine that does substring:

```
sub substr {
    my ($what, $where, $howmuch) = @_;
    unpack("x$where a$howmuch", $what);
}
```

and then there's

```
sub ordinal { unpack("W",$_[0]); } # same as ord()
```

In addition to fields allowed in `pack`, you may prefix a field with a %<number> to indicate that you want a <number>-bit checksum of the items instead of the items themselves. Default is a 16-bit checksum. The checksum is calculated by summing numeric values of expanded values (for string fields the sum of `ord($char)` is taken; for bit fields the sum of zeroes and ones).

For example, the following computes the same number as the System V sum program:

```
my $checksum = do {
    local $/;  # slurp!
    unpack("%32W*", readline) % 65535;
};
```

The following efficiently counts the number of set bits in a bit vector:

```
my $setbits = unpack("%32b*", $selectmask);
```

The `p` and `P` formats should be used with care. Since Perl has no way of checking whether the value passed to `unpack` corresponds to a valid memory location, passing a pointer value that's not known to be valid is likely to have disastrous consequences.

If there are more pack codes or if the repeat count of a field or a group is larger than what the remainder of the input string allows, the result is not well defined: the repeat count may be decreased, or `unpack` may produce empty strings or zeros, or it may raise an exception. If the input string is longer than one described by the TEMPLATE, the remainder of that input string is ignored.

See `pack` for more examples and notes.

unshift ARRAY,LIST

Does the opposite of a `shift`. Or the opposite of a `push`, depending on how you look at it. Prepends list to the front of the array and returns the new number of elements in the array.

```
unshift(@ARGV, '-e') unless $ARGV[0] =~ /^-/;
```

Note the LIST is prepended whole, not one element at a time, so the prepended elements stay in the same order. Use `reverse` to do the reverse.

Starting with Perl 5.14, an experimental feature allowed `unshift` to take a scalar expression. This experiment has been deemed unsuccessful, and was removed as of Perl 5.24.

untie VARIABLE

Breaks the binding between a variable and a package. (See tie.) Has no effect if the variable is not tied.

use Module VERSION LIST
use Module VERSION
use Module LIST
use Module
use VERSION

Imports some semantics into the current package from the named module, generally by aliasing certain subroutine or variable names into your package. It is exactly equivalent to

```
BEGIN { require Module; Module->import( LIST ); }
```

except that Module *must* be a bareword. The importation can be made conditional by using the if module.

In the peculiar `use VERSION` form, VERSION may be either a positive decimal fraction such as 5.006, which will be compared to `$]`, or a v-string of the form v5.6.1, which will be compared to `$^V` (aka `$PERL_VERSION`). An exception is raised if VERSION is greater than the version of the current

Perl interpreter; Perl will not attempt to parse the rest of the file. Compare with `require`, which can do a similar check at run time. Symmetrically, `no VERSION` allows you to specify that you want a version of Perl older than the specified one.

Specifying VERSION as a literal of the form v5.6.1 should generally be avoided, because it leads to misleading error messages under earlier versions of Perl (that is, prior to 5.6.0) that do not support this syntax. The equivalent numeric version should be used instead.

```
use v5.6.1;     # compile time version check
use 5.6.1;      # ditto
use 5.006_001;  # ditto; preferred for backwards compatibility
```

This is often useful if you need to check the current Perl version before `use`ing library modules that won't work with older versions of Perl. (We try not to do this more than we have to.)

`use VERSION` also lexically enables all features available in the requested version as defined by the feature pragma, disabling any features not in the requested version's feature bundle. See feature. Similarly, if the specified Perl version is greater than or equal to 5.12.0, strictures are enabled lexically as with `use strict`. Any explicit use of `use strict` or `no strict` overrides `use VERSION`, even if it comes before it. Later use of `use VERSION` will override all behavior of a previous `use VERSION`, possibly removing the `strict` and `feature` added by `use VERSION`. `use VERSION` does not load the *feature.pm* or *strict.pm* files.

The `BEGIN` forces the `require` and `import` to happen at compile time. The `require` makes sure the module is loaded into memory if it hasn't been yet. The `import` is not a builtin; it's just an ordinary static method call into the `Module` package to tell the module to import the list of features back into the current package. The module can implement its `import` method any way it likes, though most modules just choose to derive their `import` method via inheritance from the `Exporter` class that is defined in the `Exporter` module. See Exporter. If no `import` method can be found, then the call is skipped, even if there is an AUTOLOAD method.

If you do not want to call the package's `import` method (for instance, to stop your namespace from being altered), explicitly supply the empty list:

```
use Module ();
```

That is exactly equivalent to

```
BEGIN { require Module }
```

If the VERSION argument is present between Module and LIST, then the `use` will call the `VERSION` method in class Module with the given version as an argument:

```
use Module 12.34;
```

is equivalent to:

```
BEGIN { require Module; Module->VERSION(12.34) }
```

The default `VERSION` method, inherited from the `UNIVERSAL` class, croaks if the given version is larger than the value of the variable `$Module::VERSION`.

Again, there is a distinction between omitting LIST (`import` called with no arguments) and an explicit empty LIST `()` (`import` not called). Note that there is no comma after VERSION!

Because this is a wide-open interface, pragmas (compiler directives) are also implemented this way. Some of the currently implemented pragmas are:

```
use constant;
use diagnostics;
use integer;
use sigtrap  qw(SEGV BUS);
use strict   qw(subs vars refs);
use subs     qw(afunc blurfl);
use warnings qw(all);
use sort     qw(stable _quicksort _mergesort);
```

Some of these pseudo-modules import semantics into the current block scope (like `strict` or

`integer`, unlike ordinary modules, which import symbols into the current package (which are effective through the end of the file).

Because `use` takes effect at compile time, it doesn't respect the ordinary flow control of the code being compiled. In particular, putting a `use` inside the false branch of a conditional doesn't prevent it from being processed. If a module or pragma only needs to be loaded conditionally, this can be done using the if pragma:

```
use if $] < 5.008, "utf8";
use if WANT_WARNINGS, warnings => qw(all);
```

There's a corresponding `no` declaration that unimports meanings imported by `use`, i.e., it calls `Module->unimport(LIST)` instead of `import`. It behaves just as `import` does with VERSION, an omitted or empty LIST, or no unimport method being found.

```
no integer;
no strict 'refs';
no warnings;
```

Care should be taken when using the `no VERSION` form of `no`. It is *only* meant to be used to assert that the running Perl is of a earlier version than its argument and *not* to undo the feature-enabling side effects of `use VERSION`.

See perlmodlib for a list of standard modules and pragmas. See perlrun for the `-M` and `-m` command-line options to Perl that give `use` functionality from the command-line.

utime LIST

Changes the access and modification times on each file of a list of files. The first two elements of the list must be the NUMERIC access and modification times, in that order. Returns the number of files successfully changed. The inode change time of each file is set to the current time. For example, this code has the same effect as the Unix *touch*(1) command when the files *already exist* and belong to the user running the program:

```
#!/usr/bin/perl
my $atime = my $mtime = time;
utime $atime, $mtime, @ARGV;
```

Since Perl 5.8.0, if the first two elements of the list are `undef`, the *utime*(2) syscall from your C library is called with a null second argument. On most systems, this will set the file's access and modification times to the current time (i.e., equivalent to the example above) and will work even on files you don't own provided you have write permission:

```
for my $file (@ARGV) {
    utime(undef, undef, $file)
        || warn "Couldn't touch $file: $!";
}
```

Under NFS this will use the time of the NFS server, not the time of the local machine. If there is a time synchronization problem, the NFS server and local machine will have different times. The Unix *touch*(1) command will in fact normally use this form instead of the one shown in the first example.

Passing only one of the first two elements as `undef` is equivalent to passing a 0 and will not have the effect described when both are `undef`. This also triggers an uninitialized warning.

On systems that support *futimes*(2), you may pass filehandles among the files. On systems that don't support *futimes*(2), passing filehandles raises an exception. Filehandles must be passed as globs or glob references to be recognized; barewords are considered filenames.

Portability issues: "utime" in perlport.

values HASH

values ARRAY

In list context, returns a list consisting of all the values of the named hash. In Perl 5.12 or later only, will also return a list of the values of an array; prior to that release, attempting to use an array argument will produce a syntax error. In scalar context, returns the number of values.

Hash entries are returned in an apparently random order. The actual random order is specific to a

given hash; the exact same series of operations on two hashes may result in a different order for each hash. Any insertion into the hash may change the order, as will any deletion, with the exception that the most recent key returned by `each` or `keys` may be deleted without changing the order. So long as a given hash is unmodified you may rely on `keys`, `values` and `each` to repeatedly return the same order as each other. See "Algorithmic Complexity Attacks" in perlsec for details on why hash order is randomized. Aside from the guarantees provided here the exact details of Perl's hash algorithm and the hash traversal order are subject to change in any release of Perl. Tied hashes may behave differently to Perl's hashes with respect to changes in order on insertion and deletion of items.

As a side effect, calling `values` resets the HASH or ARRAY's internal iterator, see `each`. (In particular, calling `values` in void context resets the iterator with no other overhead. Apart from resetting the iterator, `values @array` in list context is the same as plain `@array`. (We recommend that you use void context `keys @array` for this, but reasoned that taking `values @array` out would require more documentation than leaving it in.)

Note that the values are not copied, which means modifying them will modify the contents of the hash:

```
for (values %hash)      { s/foo/bar/g }  # modifies %hash values
for (@hash{keys %hash}) { s/foo/bar/g }  # same
```

Starting with Perl 5.14, an experimental feature allowed `values` to take a scalar expression. This experiment has been deemed unsuccessful, and was removed as of Perl 5.24.

To avoid confusing would-be users of your code who are running earlier versions of Perl with mysterious syntax errors, put this sort of thing at the top of your file to signal that your code will work *only* on Perls of a recent vintage:

```
use 5.012;  # so keys/values/each work on arrays
```

See also `keys`, `each`, and `sort`.

vec EXPR,OFFSET,BITS

Treats the string in EXPR as a bit vector made up of elements of width BITS and returns the value of the element specified by OFFSET as an unsigned integer. BITS therefore specifies the number of bits that are reserved for each element in the bit vector. This must be a power of two from 1 to 32 (or 64, if your platform supports that).

If BITS is 8, "elements" coincide with bytes of the input string.

If BITS is 16 or more, bytes of the input string are grouped into chunks of size BITS/8, and each group is converted to a number as with `pack`/`unpack` with big-endian formats `n`/`N` (and analogously for BITS==64). See `pack` for details.

If bits is 4 or less, the string is broken into bytes, then the bits of each byte are broken into 8/BITS groups. Bits of a byte are numbered in a little-endian-ish way, as in `0x01`, `0x02`, `0x04`, `0x08`, `0x10`, `0x20`, `0x40`, `0x80`. For example, breaking the single input byte `chr(0x36)` into two groups gives a list `(0x6, 0x3)`; breaking it into 4 groups gives `(0x2, 0x1, 0x3, 0x0)`.

`vec` may also be assigned to, in which case parentheses are needed to give the expression the correct precedence as in

```
vec($image, $max_x * $x + $y, 8) = 3;
```

If the selected element is outside the string, the value 0 is returned. If an element off the end of the string is written to, Perl will first extend the string with sufficiently many zero bytes. It is an error to try to write off the beginning of the string (i.e., negative OFFSET).

If the string happens to be encoded as UTF-8 internally (and thus has the UTF8 flag set), `vec` tries to convert it to use a one-byte-per-character internal representation. However, if the string contains characters with values of 256 or higher, that conversion will fail. In that situation, `vec` will operate on the underlying buffer regardless, in its internal UTF-8 representation.

Strings created with `vec` can also be manipulated with the logical operators `|`, `&`, `^`, and `~`. These operators will assume a bit vector operation is desired when both operands are strings. See "Bitwise String Operators" in perlop.

The following code will build up an ASCII string saying `'PerlPerlPerl'`. The comments show

the string after each step. Note that this code works in the same way on big-endian or little-endian machines.

```
my $foo = '';
vec($foo,  0, 32) = 0x5065726C; # 'Perl'

# $foo eq "Perl" eq "\x50\x65\x72\x6C", 32 bits
print vec($foo, 0, 8);  # prints 80 == 0x50 == ord('P')

vec($foo,  2, 16) = 0x5065; # 'PerlPe'
vec($foo,  3, 16) = 0x726C; # 'PerlPerl'
vec($foo,  8,  8) = 0x50;   # 'PerlPerlP'
vec($foo,  9,  8) = 0x65;   # 'PerlPerlPe'
vec($foo, 20,  4) = 2;      # 'PerlPerlPe'   . "\x02"
vec($foo, 21,  4) = 7;      # 'PerlPerlPer'
                               # 'r' is "\x72"
vec($foo, 45,  2) = 3;      # 'PerlPerlPer'  . "\x0c"
vec($foo, 93,  1) = 1;      # 'PerlPerlPer'  . "\x2c"
vec($foo, 94,  1) = 1;      # 'PerlPerlPerl'
                               # 'l' is "\x6c"
```

To transform a bit vector into a string or list of 0's and 1's, use these:

```
my $bits = unpack("b*", $vector);
my @bits = split(//, unpack("b*", $vector));
```

If you know the exact length in bits, it can be used in place of the `*`.

Here is an example to illustrate how the bits actually fall in place:

```
#!/usr/bin/perl -wl

print <<'EOT';
                                    0         1         2         3
                     unpack("V",$_) 01234567890123456789012345678901
------------------------------------------------------------------
EOT

for $w (0..3) {
    $width = 2**$w;
    for ($shift=0; $shift < $width; ++$shift) {
        for ($off=0; $off < 32/$width; ++$off) {
            $str = pack("B*", "0"x32);
            $bits = (1<<$shift);
            vec($str, $off, $width) = $bits;
            $res = unpack("b*",$str);
            $val = unpack("V", $str);
            write;
        }
    }
}

format STDOUT =
vec($_,@#,@#) = @<< == @######### @>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
$off, $width, $bits, $val, $res
.
__END__
```

Regardless of the machine architecture on which it runs, the example above should print the following table:

```
                                        0         1         2         3
                           unpack("V",$_) 01234567890123456789012345678901
------------------------------------------------------------------------
vec($_, 0, 1) = 1   ==           1 10000000000000000000000000000000
vec($_, 1, 1) = 1   ==           2 01000000000000000000000000000000
vec($_, 2, 1) = 1   ==           4 00100000000000000000000000000000
vec($_, 3, 1) = 1   ==           8 00010000000000000000000000000000
vec($_, 4, 1) = 1   ==          16 00001000000000000000000000000000
vec($_, 5, 1) = 1   ==          32 00000100000000000000000000000000
vec($_, 6, 1) = 1   ==          64 00000010000000000000000000000000
vec($_, 7, 1) = 1   ==         128 00000001000000000000000000000000
vec($_, 8, 1) = 1   ==         256 00000000100000000000000000000000
vec($_, 9, 1) = 1   ==         512 00000000010000000000000000000000
vec($_,10, 1) = 1   ==        1024 00000000001000000000000000000000
vec($_,11, 1) = 1   ==        2048 00000000000100000000000000000000
vec($_,12, 1) = 1   ==        4096 00000000000010000000000000000000
vec($_,13, 1) = 1   ==        8192 00000000000001000000000000000000
vec($_,14, 1) = 1   ==       16384 00000000000000100000000000000000
vec($_,15, 1) = 1   ==       32768 00000000000000010000000000000000
vec($_,16, 1) = 1   ==       65536 00000000000000001000000000000000
vec($_,17, 1) = 1   ==      131072 00000000000000000100000000000000
vec($_,18, 1) = 1   ==      262144 00000000000000000010000000000000
vec($_,19, 1) = 1   ==      524288 00000000000000000001000000000000
vec($_,20, 1) = 1   ==     1048576 00000000000000000000100000000000
vec($_,21, 1) = 1   ==     2097152 00000000000000000000010000000000
vec($_,22, 1) = 1   ==     4194304 00000000000000000000001000000000
vec($_,23, 1) = 1   ==     8388608 00000000000000000000000100000000
vec($_,24, 1) = 1   ==    16777216 00000000000000000000000010000000
vec($_,25, 1) = 1   ==    33554432 00000000000000000000000001000000
vec($_,26, 1) = 1   ==    67108864 00000000000000000000000000100000
vec($_,27, 1) = 1   ==   134217728 00000000000000000000000000010000
vec($_,28, 1) = 1   ==   268435456 00000000000000000000000000001000
vec($_,29, 1) = 1   ==   536870912 00000000000000000000000000000100
vec($_,30, 1) = 1   == 1073741824 00000000000000000000000000000010
vec($_,31, 1) = 1   == 2147483648 00000000000000000000000000000001
vec($_, 0, 2) = 1   ==           1 10000000000000000000000000000000
vec($_, 1, 2) = 1   ==           4 00100000000000000000000000000000
vec($_, 2, 2) = 1   ==          16 00001000000000000000000000000000
vec($_, 3, 2) = 1   ==          64 00000010000000000000000000000000
vec($_, 4, 2) = 1   ==         256 00000000100000000000000000000000
vec($_, 5, 2) = 1   ==        1024 00000000001000000000000000000000
vec($_, 6, 2) = 1   ==        4096 00000000000010000000000000000000
vec($_, 7, 2) = 1   ==       16384 00000000000000100000000000000000
vec($_, 8, 2) = 1   ==       65536 00000000000000001000000000000000
vec($_, 9, 2) = 1   ==      262144 00000000000000000010000000000000
vec($_,10, 2) = 1   ==     1048576 00000000000000000000100000000000
vec($_,11, 2) = 1   ==     4194304 00000000000000000000001000000000
vec($_,12, 2) = 1   ==    16777216 00000000000000000000000010000000
vec($_,13, 2) = 1   ==    67108864 00000000000000000000000000100000
vec($_,14, 2) = 1   ==   268435456 00000000000000000000000000001000
vec($_,15, 2) = 1   == 1073741824 00000000000000000000000000000010
vec($_, 0, 2) = 2   ==           2 01000000000000000000000000000000
vec($_, 1, 2) = 2   ==           8 00010000000000000000000000000000
vec($_, 2, 2) = 2   ==          32 00000100000000000000000000000000
vec($_, 3, 2) = 2   ==         128 00000001000000000000000000000000
vec($_, 4, 2) = 2   ==         512 00000000010000000000000000000000
vec($_, 5, 2) = 2   ==        2048 00000000000100000000000000000000
vec($_, 6, 2) = 2   ==        8192 00000000000001000000000000000000
vec($_, 7, 2) = 2   ==       32768 00000000000000010000000000000000
```

```
vec($_, 8, 2) = 2   ==     131072 00000000000000000100000000000000
vec($_, 9, 2) = 2   ==     524288 00000000000000000001000000000000
vec($_,10, 2) = 2   ==    2097152 00000000000000000000010000000000
vec($_,11, 2) = 2   ==    8388608 00000000000000000000000100000000
vec($_,12, 2) = 2   ==   33554432 00000000000000000000000001000000
vec($_,13, 2) = 2   ==  134217728 00000000000000000000000000010000
vec($_,14, 2) = 2   ==  536870912 00000000000000000000000000000100
vec($_,15, 2) = 2   == 2147483648 00000000000000000000000000000001
vec($_, 0, 4) = 1   ==          1 10000000000000000000000000000000
vec($_, 1, 4) = 1   ==         16 00001000000000000000000000000000
vec($_, 2, 4) = 1   ==        256 00000001000000000000000000000000
vec($_, 3, 4) = 1   ==       4096 00000000000010000000000000000000
vec($_, 4, 4) = 1   ==      65536 00000000000000010000000000000000
vec($_, 5, 4) = 1   ==    1048576 00000000000000000001000000000000
vec($_, 6, 4) = 1   ==   16777216 00000000000000000000000010000000
vec($_, 7, 4) = 1   ==  268435456 00000000000000000000000000001000
vec($_, 0, 4) = 2   ==          2 01000000000000000000000000000000
vec($_, 1, 4) = 2   ==         32 00000100000000000000000000000000
vec($_, 2, 4) = 2   ==        512 00000000100000000000000000000000
vec($_, 3, 4) = 2   ==       8192 00000000000001000000000000000000
vec($_, 4, 4) = 2   ==     131072 00000000000000010000000000000000
vec($_, 5, 4) = 2   ==    2097152 00000000000000000000010000000000
vec($_, 6, 4) = 2   ==   33554432 00000000000000000000000001000000
vec($_, 7, 4) = 2   ==  536870912 00000000000000000000000000000100
vec($_, 0, 4) = 4   ==          4 00100000000000000000000000000000
vec($_, 1, 4) = 4   ==         64 00000100000000000000000000000000
vec($_, 2, 4) = 4   ==       1024 00000000001000000000000000000000
vec($_, 3, 4) = 4   ==      16384 00000000000000100000000000000000
vec($_, 4, 4) = 4   ==     262144 00000000000000000010000000000000
vec($_, 5, 4) = 4   ==    4194304 00000000000000000000001000000000
vec($_, 6, 4) = 4   ==   67108864 00000000000000000000000000100000
vec($_, 7, 4) = 4   == 1073741824 00000000000000000000000000000010
vec($_, 0, 4) = 8   ==          8 00010000000000000000000000000000
vec($_, 1, 4) = 8   ==        128 00000001000000000000000000000000
vec($_, 2, 4) = 8   ==       2048 00000000000100000000000000000000
vec($_, 3, 4) = 8   ==      32768 00000000000000010000000000000000
vec($_, 4, 4) = 8   ==     524288 00000000000000000001000000000000
vec($_, 5, 4) = 8   ==    8388608 00000000000000000000000100000000
vec($_, 6, 4) = 8   ==  134217728 00000000000000000000000000010000
vec($_, 7, 4) = 8   == 2147483648 00000000000000000000000000000001
vec($_, 0, 8) = 1   ==          1 10000000000000000000000000000000
vec($_, 1, 8) = 1   ==        256 00000000100000000000000000000000
vec($_, 2, 8) = 1   ==      65536 00000000000000000100000000000000
vec($_, 3, 8) = 1   ==   16777216 00000000000000000000000010000000
vec($_, 0, 8) = 2   ==          2 01000000000000000000000000000000
vec($_, 1, 8) = 2   ==        512 00000000100000000000000000000000
vec($_, 2, 8) = 2   ==     131072 00000000000000000100000000000000
vec($_, 3, 8) = 2   ==   33554432 00000000000000000000000001000000
vec($_, 0, 8) = 4   ==          4 00100000000000000000000000000000
vec($_, 1, 8) = 4   ==       1024 00000000001000000000000000000000
vec($_, 2, 8) = 4   ==     262144 00000000000000000010000000000000
vec($_, 3, 8) = 4   ==   67108864 00000000000000000000000000100000
vec($_, 0, 8) = 8   ==          8 00010000000000000000000000000000
vec($_, 1, 8) = 8   ==       2048 00000000000100000000000000000000
vec($_, 2, 8) = 8   ==     524288 00000000000000000001000000000000
vec($_, 3, 8) = 8   ==  134217728 00000000000000000000000000010000
vec($_, 0, 8) = 16  ==         16 00001000000000000000000000000000
vec($_, 1, 8) = 16  ==       4096 00000000000010000000000000000000
vec($_, 2, 8) = 16  ==    1048576 00000000000000000001000000000000
```

```
vec($_, 3, 8) = 16  ==  268435456 00000000000000000000000000001000
vec($_, 0, 8) = 32  ==         32 00000100000000000000000000000000
vec($_, 1, 8) = 32  ==       8192 00000000000001000000000000000000
vec($_, 2, 8) = 32  ==    2097152 00000000000000000000010000000000
vec($_, 3, 8) = 32  ==  536870912 00000000000000000000000000000100
vec($_, 0, 8) = 64  ==         64 00000100000000000000000000000000
vec($_, 1, 8) = 64  ==      16384 00000000000000010000000000000000
vec($_, 2, 8) = 64  ==    4194304 00000000000000000000001000000000
vec($_, 3, 8) = 64  == 1073741824 00000000000000000000000000000010
vec($_, 0, 8) = 128 ==        128 00000001000000000000000000000000
vec($_, 1, 8) = 128 ==      32768 00000000000000010000000000000000
vec($_, 2, 8) = 128 ==    8388608 00000000000000000000000100000000
vec($_, 3, 8) = 128 == 2147483648 00000000000000000000000000000001
```

wait

Behaves like *wait*(2) on your system: it waits for a child process to terminate and returns the pid of the deceased process, or `-1` if there are no child processes. The status is returned in `$?` and `${^CHILD_ERROR_NATIVE}`. Note that a return value of `-1` could mean that child processes are being automatically reaped, as described in perlipc.

If you use `wait` in your handler for `$SIG{CHLD}`, it may accidentally wait for the child created by `qx` or `system`. See perlipc for details.

Portability issues: "wait" in perlport.

waitpid PID,FLAGS

Waits for a particular child process to terminate and returns the pid of the deceased process, or `-1` if there is no such child process. A non-blocking wait (with WNOHANG in FLAGS) can return 0 if there are child processes matching PID but none have terminated yet. The status is returned in `$?` and `${^CHILD_ERROR_NATIVE}`.

A PID of `0` indicates to wait for any child process whose process group ID is equal to that of the current process. A PID of less than `-1` indicates to wait for any child process whose process group ID is equal to −PID. A PID of `-1` indicates to wait for any child process.

If you say

```
use POSIX ":sys_wait_h";

my $kid;
do {
    $kid = waitpid(-1, WNOHANG);
} while $kid > 0;
```

or

```
1 while waitpid(-1, WNOHANG) > 0;
```

then you can do a non-blocking wait for all pending zombie processes (see "WAIT" in POSIX). Non-blocking wait is available on machines supporting either the *waitpid*(2) or *wait4*(2) syscalls. However, waiting for a particular pid with FLAGS of `0` is implemented everywhere. (Perl emulates the system call by remembering the status values of processes that have exited but have not been harvested by the Perl script yet.)

Note that on some systems, a return value of `-1` could mean that child processes are being automatically reaped. See perlipc for details, and for other examples.

Portability issues: "waitpid" in perlport.

wantarray

Returns true if the context of the currently executing subroutine or `eval` is looking for a list value. Returns false if the context is looking for a scalar. Returns the undefined value if the context is looking for no value (void context).

```
return unless defined wantarray; # don't bother doing more
my @a = complex_calculation();
return wantarray ? @a : "@a";
```

`wantarray`'s result is unspecified in the top level of a file, in a `BEGIN`, `UNITCHECK`, `CHECK`, `INIT` or `END` block, or in a `DESTROY` method.

This function should have been named *wantlist()* instead.

warn LIST

Prints the value of LIST to STDERR. If the last element of LIST does not end in a newline, it appends the same file/line number text as `die` does.

If the output is empty and `$@` already contains a value (typically from a previous eval) that value is used after appending `"\t...caught"` to `$@`. This is useful for staying almost, but not entirely similar to `die`.

If `$@` is empty, then the string `"Warning: Something's wrong"` is used.

No message is printed if there is a `$SIG{__WARN__}` handler installed. It is the handler's responsibility to deal with the message as it sees fit (like, for instance, converting it into a `die`). Most handlers must therefore arrange to actually display the warnings that they are not prepared to deal with, by calling `warn` again in the handler. Note that this is quite safe and will not produce an endless loop, since `__WARN__` hooks are not called from inside one.

You will find this behavior is slightly different from that of `$SIG{__DIE__}` handlers (which don't suppress the error text, but can instead call `die` again to change it).

Using a `__WARN__` handler provides a powerful way to silence all warnings (even the so-called mandatory ones). An example:

```
# wipe out *all* compile-time warnings
BEGIN { $SIG{'__WARN__'} = sub { warn $_[0] if $DOWARN } }
my $foo = 10;
my $foo = 20;          # no warning about duplicate my $foo,
                       # but hey, you asked for it!
# no compile-time or run-time warnings before here
$DOWARN = 1;

# run-time warnings enabled after here
warn "\$foo is alive and $foo!";     # does show up
```

See perlvar for details on setting `%SIG` entries and for more examples. See the Carp module for other kinds of warnings using its `carp` and `cluck` functions.

write FILEHANDLE
write EXPR
write

Writes a formatted record (possibly multi-line) to the specified FILEHANDLE, using the format associated with that file. By default the format for a file is the one having the same name as the filehandle, but the format for the current output channel (see the `select` function) may be set explicitly by assigning the name of the format to the `$~` variable.

Top of form processing is handled automatically: if there is insufficient room on the current page for the formatted record, the page is advanced by writing a form feed and a special top-of-page format is used to format the new page header before the record is written. By default, the top-of-page format is the name of the filehandle with `_TOP` appended, or `top` in the current package if the former does not exist. This would be a problem with autovivified filehandles, but it may be dynamically set to the format of your choice by assigning the name to the `$^` variable while that filehandle is selected. The number of lines remaining on the current page is in variable `$-`, which can be set to `0` to force a new page.

If FILEHANDLE is unspecified, output goes to the current default output channel, which starts out as STDOUT but may be changed by the `select` operator. If the FILEHANDLE is an EXPR, then the expression is evaluated and the resulting string is used to look up the name of the FILEHANDLE at run

time. For more on formats, see perlform.

Note that write is *not* the opposite of `read`. Unfortunately.

y/// The transliteration operator. Same as `tr///`. See ''Quote-Like Operators'' in perlop.

Non-function Keywords by Cross-reference

perldata

__DATA__
__END__
These keywords are documented in ''Special Literals'' in perldata.

perlmod

BEGIN
CHECK
END
INIT
UNITCHECK
These compile phase keywords are documented in ''BEGIN, UNITCHECK, CHECK, INIT and END'' in perlmod.

perlobj

DESTROY
This method keyword is documented in ''Destructors'' in perlobj.

perlop

and
cmp
eq
ge
gt
le
lt
ne
not
or
x
xor These operators are documented in perlop.

perlsub

AUTOLOAD
This keyword is documented in ''Autoloading'' in perlsub.

perlsyn

else
elsif
for
foreach
if
unless
until
while
These flow-control keywords are documented in ''Compound Statements'' in perlsyn.

elseif
The ''else if'' keyword is spelled `elsif` in Perl. There's no `elif` or `else if` either. It does parse `elseif`, but only to warn you about not using it.

See the documentation for flow-control keywords in ''Compound Statements'' in perlsyn.

default

given
when
: These flow-control keywords related to the experimental switch feature are documented in ‘‘Switch Statements’’ in perlsyn.

NAME

perlvar – Perl predefined variables

DESCRIPTION

The Syntax of Variable Names

Variable names in Perl can have several formats. Usually, they must begin with a letter or underscore, in which case they can be arbitrarily long (up to an internal limit of 251 characters) and may contain letters, digits, underscores, or the special sequence `::` or `'`. In this case, the part before the last `::` or `'` is taken to be a *package qualifier*; see perlmod. A Unicode letter that is not ASCII is not considered to be a letter unless `"use utf8"` is in effect, and somewhat more complicated rules apply; see "Identifier parsing" in perldata for details.

Perl variable names may also be a sequence of digits, a single punctuation character, or the two-character sequence: `^` (caret or CIRCUMFLEX ACCENT) followed by any one of the characters `[][A-Z^_?\]`. These names are all reserved for special uses by Perl; for example, the all-digits names are used to hold data captured by backreferences after a regular expression match.

Since Perl v5.6.0, Perl variable names may also be alphanumeric strings preceded by a caret. These must all be written in the form `${^Foo}`; the braces are not optional. `${^Foo}` denotes the scalar variable whose name is considered to be a control-`F` followed by two `o`'s. These variables are reserved for future special uses by Perl, except for the ones that begin with `^_` (caret-underscore). No name that begins with `^_` will acquire a special meaning in any future version of Perl; such names may therefore be used safely in programs. `$^_` itself, however, *is* reserved.

Perl identifiers that begin with digits or punctuation characters are exempt from the effects of the `package` declaration and are always forced to be in package `main`; they are also exempt from `strict 'vars'` errors. A few other names are also exempt in these ways:

```
ENV       STDIN
INC       STDOUT
ARGV      STDERR
ARGVOUT
SIG
```

In particular, the special `${^_XYZ}` variables are always taken to be in package `main`, regardless of any `package` declarations presently in scope.

SPECIAL VARIABLES

The following names have special meaning to Perl. Most punctuation names have reasonable mnemonics, or analogs in the shells. Nevertheless, if you wish to use long variable names, you need only say:

```
use English;
```

at the top of your program. This aliases all the short names to the long names in the current package. Some even have medium names, generally borrowed from **awk**. For more info, please see English.

Before you continue, note the sort order for variables. In general, we first list the variables in case-insensitive, almost-lexigraphical order (ignoring the `{` or `^` preceding words, as in `${^UNICODE}` or `$^T`), although `$_` and `@_` move up to the top of the pile. For variables with the same identifier, we list it in order of scalar, array, hash, and bareword.

General Variables

`$ARG`

`$_` The default input and pattern-searching space. The following pairs are equivalent:

```
while (<>) {...}    # equivalent only in while!
while (defined($_ = <>)) {...}

/^Subject:/
$_ =~ /^Subject:/

tr/a-z/A-Z/
$_ =~ tr/a-z/A-Z/

chomp
chomp($_)
```

Here are the places where Perl will assume `$_` even if you don't use it:

- The following functions use `$_` as a default argument:

  abs, alarm, chomp, chop, chr, chroot, cos, defined, eval, evalbytes, exp, fc, glob, hex, int, lc, lcfirst, length, log, lstat, mkdir, oct, ord, pos, print, printf, quotemeta, readlink, readpipe, ref, require, reverse (in scalar context only), rmdir, say, sin, split (for its second argument), sqrt, stat, study, uc, ucfirst, unlink, unpack.

- All file tests (`-f`, `-d`) except for `-t`, which defaults to STDIN. See "-X" in perlfunc
- The pattern matching operations `m//`, `s///` and `tr///` (aka `y///`) when used without an `=~` operator.
- The default iterator variable in a `foreach` loop if no other variable is supplied.
- The implicit iterator variable in the `grep()` and `map()` functions.
- The implicit variable of `given()`.
- The default place to put the next value or input record when a `<FH>`, `readline`, `readdir` or `each` operation's result is tested by itself as the sole criterion of a `while` test. Outside a `while` test, this will not happen.

`$_` is by default a global variable. However, as of perl v5.10.0, you can use a lexical version of `$_` by declaring it in a file or in a block with `my`. Moreover, declaring `our $_` restores the global `$_` in the current scope. Though this seemed like a good idea at the time it was introduced, lexical `$_` actually causes more problems than it solves. If you call a function that expects to be passed information via `$_`, it may or may not work, depending on how the function is written, there not being any easy way to solve this. Just avoid lexical `$_`, unless you are feeling particularly masochistic. For this reason lexical `$_` is still experimental and will produce a warning unless warnings have been disabled. As with other experimental features, the behavior of lexical `$_` is subject to change without notice, including change into a fatal error.

Mnemonic: underline is understood in certain operations.

`@ARG`

`@_` Within a subroutine the array `@_` contains the parameters passed to that subroutine. Inside a subroutine, `@_` is the default array for the array operators `pop` and `shift`.

See perlsub.

`$LIST_SEPARATOR`

`$"` When an array or an array slice is interpolated into a double-quoted string or a similar context such as `/.../`, its elements are separated by this value. Default is a space. For example, this:

```
print "The array is: @array\n";
```

is equivalent to this:

```
print "The array is: " . join($", @array) . "\n";
```

Mnemonic: works in double-quoted context.

`$PROCESS_ID`

`$PID`

`$$` The process number of the Perl running this script. Though you *can* set this variable, doing so is generally discouraged, although it can be invaluable for some testing purposes. It will be reset automatically across `fork()` calls.

Note for Linux and Debian GNU/kFreeBSD users: Before Perl v5.16.0 perl would emulate POSIX semantics on Linux systems using LinuxThreads, a partial implementation of POSIX Threads that has since been superseded by the Native POSIX Thread Library (NPTL).

LinuxThreads is now obsolete on Linux, and caching `getpid()` like this made embedding perl unnecessarily complex (since you'd have to manually update the value of $$), so now `$$` and `getppid()` will always return the same values as the underlying C library.

Debian GNU/kFreeBSD systems also used LinuxThreads up until and including the 6.0 release, but after that moved to FreeBSD thread semantics, which are POSIX-like.

To see if your system is affected by this discrepancy check if `getconf GNU_LIBPTHREAD_VERSION | grep -q NPTL` returns a false value. NTPL threads preserve the POSIX semantics.

Mnemonic: same as shells.

`$PROGRAM_NAME`

`$0` Contains the name of the program being executed.

On some (but not all) operating systems assigning to `$0` modifies the argument area that the `ps` program sees. On some platforms you may have to use special `ps` options or a different `ps` to see the changes. Modifying the `$0` is more useful as a way of indicating the current program state than it is for hiding the program you're running.

Note that there are platform-specific limitations on the maximum length of `$0`. In the most extreme case it may be limited to the space occupied by the original `$0`.

In some platforms there may be arbitrary amount of padding, for example space characters, after the modified name as shown by `ps`. In some platforms this padding may extend all the way to the original length of the argument area, no matter what you do (this is the case for example with Linux 2.2).

Note for BSD users: setting `$0` does not completely remove "perl" from the *ps*(1) output. For example, setting `$0` to `"foobar"` may result in `"perl: foobar (perl)"` (whether both the `"perl: "` prefix and the " (perl)" suffix are shown depends on your exact BSD variant and version). This is an operating system feature, Perl cannot help it.

In multithreaded scripts Perl coordinates the threads so that any thread may modify its copy of the `$0` and the change becomes visible to *ps*(1) (assuming the operating system plays along). Note that the view of `$0` the other threads have will not change since they have their own copies of it.

If the program has been given to perl via the switches `-e` or `-E`, `$0` will contain the string `"-e"`.

On Linux as of perl v5.14.0 the legacy process name will be set with `prctl(2)`, in addition to altering the POSIX name via `argv[0]` as perl has done since version 4.000. Now system utilities that read the legacy process name such as ps, top and killall will recognize the name you set when assigning to `$0`. The string you supply will be cut off at 16 bytes, this is a limitation imposed by Linux.

Mnemonic: same as **sh** and **ksh**.

`$REAL_GROUP_ID`

`$GID`

`$(` The real gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by `getgid()`, and the subsequent ones by `getgroups()`, one of which may be the same as the first number.

However, a value assigned to `$(` must be a single number used to set the real gid. So the value given by `$(` should *not* be assigned back to `$(` without being forced numeric, such as by adding zero. Note that this is different to the effective gid (`$)`) which does take a list.

You can change both the real gid and the effective gid at the same time by using `POSIX::setgid()`. Changes to `$(` require a check to `$!` to detect any possible errors after an attempted change.

Mnemonic: parentheses are used to *group* things. The real gid is the group you *left*, if you're running setgid.

`$EFFECTIVE_GROUP_ID`

`$EGID`

`$)` The effective gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by `getegid()`, and the subsequent ones by `getgroups()`, one of which may be the same as the first number.

Similarly, a value assigned to `$)` must also be a space-separated list of numbers. The first

number sets the effective gid, and the rest (if any) are passed to `setgroups()`. To get the effect of an empty list for `setgroups()`, just repeat the new effective gid; that is, to force an effective gid of 5 and an effectively empty `setgroups()` list, say `$) = "5 5"`.

You can change both the effective gid and the real gid at the same time by using `POSIX::setgid()` (use only a single numeric argument). Changes to `$)` require a check to `$!` to detect any possible errors after an attempted change.

`$<`, `$>`, `$(` and `$)` can be set only on machines that support the corresponding *set[re][ug]id()* routine. `$(` and `$)` can be swapped only on machines supporting `setregid()`.

Mnemonic: parentheses are used to *group* things. The effective gid is the group that's *right* for you, if you're running setgid.

`$REAL_USER_ID`
`$UID`
`$<` The real uid of this process. You can change both the real uid and the effective uid at the same time by using `POSIX::setuid()`. Since changes to `$<` require a system call, check `$!` after a change attempt to detect any possible errors.

Mnemonic: it's the uid you came *from*, if you're running setuid.

`$EFFECTIVE_USER_ID`
`$EUID`
`$>` The effective uid of this process. For example:

```
$< = $>;            # set real to effective uid
($<,$>) = ($>,$<);  # swap real and effective uids
```

You can change both the effective uid and the real uid at the same time by using `POSIX::setuid()`. Changes to `$>` require a check to `$!` to detect any possible errors after an attempted change.

`$<` and `$>` can be swapped only on machines supporting `setreuid()`.

Mnemonic: it's the uid you went *to*, if you're running setuid.

`$SUBSCRIPT_SEPARATOR`
`$SUBSEP`
`$;` The subscript separator for multidimensional array emulation. If you refer to a hash element as

```
$foo{$x,$y,$z}
```

it really means

```
$foo{join($;, $x, $y, $z)}
```

But don't put

```
@foo{$x,$y,$z}      # a slice--note the @
```

which means

```
($foo{$x},$foo{$y},$foo{$z})
```

Default is "\034", the same as SUBSEP in **awk**. If your keys contain binary data there might not be any safe value for `$;`.

Consider using "real" multidimensional arrays as described in perllol.

Mnemonic: comma (the syntactic subscript separator) is a semi-semicolon.

`$a`
`$b` Special package variables when using `sort()`, see "sort" in perlfunc. Because of this specialness `$a` and `$b` don't need to be declared (using `use vars`, or `our()`) even when using the `strict 'vars'` pragma. Don't lexicalize them with `my $a` or `my $b` if you want to be able to use them in the `sort()` comparison block or function.

`%ENV` The hash `%ENV` contains your current environment. Setting a value in `ENV` changes the environment for any child processes you subsequently `fork()` off.

As of v5.18.0, both keys and values stored in `%ENV` are stringified.

```
my $foo = 1;
$ENV{'bar'} = \$foo;
if( ref $ENV{'bar'} ) {
    say "Pre 5.18.0 Behaviour";
} else {
    say "Post 5.18.0 Behaviour";
}
```

Previously, only child processes received stringified values:

```
my $foo = 1;
$ENV{'bar'} = \$foo;

# Always printed 'non ref'
system($^X, '-e',
       q/print ( ref $ENV{'bar'}  ? 'ref' : 'non ref' ) /);
```

This happens because you can't really share arbitrary data structures with foreign processes.

`$OLD_PERL_VERSION`

$] The revision, version, and subversion of the Perl interpreter, represented as a decimal of the form 5.XXXYYY, where XXX is the version / 1e3 and YYY is the subversion / 1e6. For example, Perl v5.10.1 would be "5.010001".

This variable can be used to determine whether the Perl interpreter executing a script is in the right range of versions:

```
warn "No PerlIO!\n" if $] lt '5.008';
```

When comparing `$]`, string comparison operators are **highly recommended**. The inherent limitations of binary floating point representation can sometimes lead to incorrect comparisons for some numbers on some architectures.

See also the documentation of `use VERSION` and `require VERSION` for a convenient way to fail if the running Perl interpreter is too old.

See "$^V" for a representation of the Perl version as a version object, which allows more flexible string comparisons.

The main advantage of `$]` over `$^V` is that it works the same on any version of Perl. The disadvantages are that it can't easily be compared to versions in other formats (e.g. literal v–strings, "v1.2.3" or version objects) and numeric comparisons can occasionally fail; it's good for string literal version checks and bad for comparing to a variable that hasn't been sanity-checked.

The `$OLD_PERL_VERSION` form was added in Perl v5.20.0 for historical reasons but its use is discouraged. (If your reason to use `$]` is to run code on old perls then referring to it as `$OLD_PERL_VERSION` would be self-defeating.)

Mnemonic: Is this version of perl in the right bracket?

`$SYSTEM_FD_MAX`

$^F The maximum system file descriptor, ordinarily 2. System file descriptors are passed to `exec()`ed processes, while higher file descriptors are not. Also, during an `open()`, system file descriptors are preserved even if the `open()` fails (ordinary file descriptors are closed before the `open()` is attempted). The close-on-exec status of a file descriptor will be decided according to the value of `$^F` when the corresponding file, pipe, or socket was opened, not the time of the `exec()`.

`@F` The array `@F` contains the fields of each line read in when autosplit mode is turned on. See perlrun for the **–a** switch. This array is package-specific, and must be declared or given a full package name if not in package main when running under `strict 'vars'`.

`@INC` The array `@INC` contains the list of places that the `do EXPR`, `require`, or `use` constructs look for their library files. It initially consists of the arguments to any **–I** command-line switches,

followed by the default Perl library, probably */usr/local/lib/perl*, followed by ".", to represent the current directory. (".” will not be appended if taint checks are enabled, either by `-T` or by `-t`, or if configured not to do so by the `-Ddefault_inc_excludes_dot` compile time option.) If you need to modify this at runtime, you should use the `use lib` pragma to get the machine-dependent library properly loaded also:

```
use lib '/mypath/libdir/';
use SomeMod;
```

You can also insert hooks into the file inclusion system by putting Perl code directly into `@INC`. Those hooks may be subroutine references, array references or blessed objects. See "require" in perlfunc for details.

`%INC` The hash `%INC` contains entries for each filename included via the `do`, `require`, or `use` operators. The key is the filename you specified (with module names converted to pathnames), and the value is the location of the file found. The `require` operator uses this hash to determine whether a particular file has already been included.

If the file was loaded via a hook (e.g. a subroutine reference, see "require" in perlfunc for a description of these hooks), this hook is by default inserted into `%INC` in place of a filename. Note, however, that the hook may have set the `%INC` entry by itself to provide some more specific info.

`$INPLACE_EDIT`

$^I The current value of the inplace-edit extension. Use `undef` to disable inplace editing.

Mnemonic: value of **–i** switch.

`@ISA` Each package contains a special array called `@ISA` which contains a list of that class's parent classes, if any. This array is simply a list of scalars, each of which is a string that corresponds to a package name. The array is examined when Perl does method resolution, which is covered in perlobj.

To load packages while adding them to `@ISA`, see the parent pragma. The discouraged base pragma does this as well, but should not be used except when compatibility with the discouraged fields pragma is required.

$^M By default, running out of memory is an untrappable, fatal error. However, if suitably built, Perl can use the contents of `$^M` as an emergency memory pool after `die()`ing. Suppose that your Perl were compiled with `-DPERL_EMERGENCY_SBRK` and used Perl's malloc. Then

```
$^M = 'a' x (1 << 16);
```

would allocate a 64K buffer for use in an emergency. See the *INSTALL* file in the Perl distribution for information on how to add custom C compilation flags when compiling perl. To discourage casual use of this advanced feature, there is no English long name for this variable.

This variable was added in Perl 5.004.

`$OSNAME`

$^O The name of the operating system under which this copy of Perl was built, as determined during the configuration process. For examples see "PLATFORMS" in perlport.

The value is identical to `$Config{'osname'}`. See also Config and the **–V** command-line switch documented in perlrun.

In Windows platforms, `$^O` is not very helpful: since it is always `MSWin32`, it doesn't tell the difference between 95/98/ME/NT/2000/XP/CE/.NET. Use `Win32::GetOSName()` or *Win32::GetOSVersion()* (see Win32 and perlport) to distinguish between the variants.

This variable was added in Perl 5.003.

`%SIG` The hash `%SIG` contains signal handlers for signals. For example:

```
sub handler {   # 1st argument is signal name
    my($sig) = @_;
    print "Caught a SIG$sig--shutting down\n";
    close(LOG);
    exit(0);
    }

$SIG{'INT'}  = \&handler;
$SIG{'QUIT'} = \&handler;
...
$SIG{'INT'}  = 'DEFAULT';   # restore default action
$SIG{'QUIT'} = 'IGNORE';    # ignore SIGQUIT
```

Using a value of `'IGNORE'` usually has the effect of ignoring the signal, except for the `CHLD` signal. See perlipc for more about this special case.

Here are some other examples:

```
$SIG{"PIPE"} = "Plumber";   # assumes main::Plumber (not
                            # recommended)
$SIG{"PIPE"} = \&Plumber;   # just fine; assume current
                            # Plumber
$SIG{"PIPE"} = *Plumber;    # somewhat esoteric
$SIG{"PIPE"} = Plumber();   # oops, what did Plumber()
                            # return??
```

Be sure not to use a bareword as the name of a signal handler, lest you inadvertently call it.

If your system has the `sigaction()` function then signal handlers are installed using it. This means you get reliable signal handling.

The default delivery policy of signals changed in Perl v5.8.0 from immediate (also known as "unsafe") to deferred, also known as "safe signals". See perlipc for more information.

Certain internal hooks can be also set using the `%SIG` hash. The routine indicated by `$SIG{__WARN__}` is called when a warning message is about to be printed. The warning message is passed as the first argument. The presence of a `__WARN__` hook causes the ordinary printing of warnings to `STDERR` to be suppressed. You can use this to save warnings in a variable, or turn warnings into fatal errors, like this:

```
local $SIG{__WARN__} = sub { die $_[0] };
eval $proggie;
```

As the `'IGNORE'` hook is not supported by `__WARN__`, you can disable warnings using the empty subroutine:

```
local $SIG{__WARN__} = sub {};
```

The routine indicated by `$SIG{__DIE__}` is called when a fatal exception is about to be thrown. The error message is passed as the first argument. When a `__DIE__` hook routine returns, the exception processing continues as it would have in the absence of the hook, unless the hook routine itself exits via a `goto &sub`, a loop exit, or a `die()`. The `__DIE__` handler is explicitly disabled during the call, so that you can die from a `__DIE__` handler. Similarly for `__WARN__`.

The `$SIG{__DIE__}` hook is called even inside an `eval()`. It was never intended to happen this way, but an implementation glitch made this possible. This used to be deprecated, as it allowed strange action at a distance like rewriting a pending exception in `$@`. Plans to rectify this have been scrapped, as users found that rewriting a pending exception is actually a useful feature, and not a bug.

`__DIE__`/`__WARN__` handlers are very special in one respect: they may be called to report (probable) errors found by the parser. In such a case the parser may be in inconsistent state, so any attempt to evaluate Perl code from such a handler will probably result in a segfault. This means that warnings or errors that result from parsing Perl should be used with extreme caution,

like this:

```
require Carp if defined $^S;
Carp::confess("Something wrong") if defined &Carp::confess;
die "Something wrong, but could not load Carp to give "
  . "backtrace...\n\t"
  . "To see backtrace try starting Perl with -MCarp switch";
```

Here the first line will load `Carp` *unless* it is the parser who called the handler. The second line will print backtrace and die if `Carp` was available. The third line will be executed only if `Carp` was not available.

Having to even think about the `$^S` variable in your exception handlers is simply wrong. `$SIG{__DIE__}` as currently implemented invites grievous and difficult to track down errors. Avoid it and use an `END{}` or CORE::GLOBAL::die override instead.

See "die" in perlfunc, "warn" in perlfunc, "eval" in perlfunc, and warnings for additional information.

`$BASETIME`
$^T The time at which the program began running, in seconds since the epoch (beginning of 1970). The values returned by the **-M**, **-A**, and **-C** filetests are based on this value.

`$PERL_VERSION`
$^V The revision, version, and subversion of the Perl interpreter, represented as a version object.

This variable first appeared in perl v5.6.0; earlier versions of perl will see an undefined value. Before perl v5.10.0 `$^V` was represented as a v-string rather than a version object.

`$^V` can be used to determine whether the Perl interpreter executing a script is in the right range of versions. For example:

```
warn "Hashes not randomized!\n" if !$^V or $^V lt v5.8.1
```

While version objects overload stringification, to portably convert `$^V` into its string representation, use `sprintf()`'s `"%vd"` conversion, which works for both v-strings or version objects:

```
printf "version is v%vd\n", $^V;  # Perl's version
```

See the documentation of `use VERSION` and `require VERSION` for a convenient way to fail if the running Perl interpreter is too old.

See also `"$]"` for a decimal representation of the Perl version.

The main advantage of `$^V` over `$]` is that, for Perl v5.10.0 or later, it overloads operators, allowing easy comparison against other version representations (e.g. decimal, literal v-string, "v1.2.3", or objects). The disadvantage is that prior to v5.10.0, it was only a literal v-string, which can't be easily printed or compared, whereas the behavior of `$]` is unchanged on all versions of Perl.

Mnemonic: use ^V for a version object.

${^WIN32_SLOPPY_STAT}
If this variable is set to a true value, then `stat()` on Windows will not try to open the file. This means that the link count cannot be determined and file attributes may be out of date if additional hardlinks to the file exist. On the other hand, not opening the file is considerably faster, especially for files on network drives.

This variable could be set in the *sitecustomize.pl* file to configure the local Perl installation to use "sloppy" `stat()` by default. See the documentation for **-f** in perlrun for more information about site customization.

This variable was added in Perl v5.10.0.

`$EXECUTABLE_NAME`
$^X The name used to execute the current copy of Perl, from C's `argv[0]` or (where supported) */proc/self/exe*.

Depending on the host operating system, the value of `$^X` may be a relative or absolute pathname of the perl program file, or may be the string used to invoke perl but not the pathname of the perl program file. Also, most operating systems permit invoking programs that are not in the PATH environment variable, so there is no guarantee that the value of `$^X` is in PATH. For VMS, the value may or may not include a version number.

You usually can use the value of `$^X` to re-invoke an independent copy of the same perl that is currently running, e.g.,

```
@first_run = `$^X -le "print int rand 100 for 1..100"`;
```

But recall that not all operating systems support forking or capturing of the output of commands, so this complex statement may not be portable.

It is not safe to use the value of `$^X` as a path name of a file, as some operating systems that have a mandatory suffix on executable files do not require use of the suffix when invoking a command. To convert the value of `$^X` to a path name, use the following statements:

```
# Build up a set of file names (not command names).
use Config;
my $this_perl = $^X;
if ($^O ne 'VMS') {
    $this_perl .= $Config{_exe}
      unless $this_perl =~ m/$Config{_exe}$/i;
    }
```

Because many operating systems permit anyone with read access to the Perl program file to make a copy of it, patch the copy, and then execute the copy, the security-conscious Perl programmer should take care to invoke the installed copy of perl, not the copy referenced by `$^X`. The following statements accomplish this goal, and produce a pathname that can be invoked as a command or referenced as a file.

```
use Config;
my $secure_perl_path = $Config{perlpath};
if ($^O ne 'VMS') {
    $secure_perl_path .= $Config{_exe}
        unless $secure_perl_path =~ m/$Config{_exe}$/i;
    }
```

Variables related to regular expressions

Most of the special variables related to regular expressions are side effects. Perl sets these variables when it has a successful match, so you should check the match result before using them. For instance:

```
if( /P(A)TT(ER)N/ ) {
    print "I found $1 and $2\n";
    }
```

These variables are read-only and dynamically-scoped, unless we note otherwise.

The dynamic nature of the regular expression variables means that their value is limited to the block that they are in, as demonstrated by this bit of code:

```
my $outer = 'Wallace and Grommit';
my $inner = 'Mutt and Jeff';

my $pattern = qr/(\S+) and (\S+)/;

sub show_n { print "\$1 is $1; \$2 is $2\n" }

{
OUTER:
    show_n() if $outer =~ m/$pattern/;

    INNER: {
        show_n() if $inner =~ m/$pattern/;
```

```
            }

        show_n();
    }
```

The output shows that while in the `OUTER` block, the values of `$1` and `$2` are from the match against `$outer`. Inside the `INNER` block, the values of `$1` and `$2` are from the match against `$inner`, but only until the end of the block (i.e. the dynamic scope). After the `INNER` block completes, the values of `$1` and `$2` return to the values for the match against `$outer` even though we have not made another match:

```
$1 is Wallace; $2 is Grommit
$1 is Mutt; $2 is Jeff
$1 is Wallace; $2 is Grommit
```

Performance issues

Traditionally in Perl, any use of any of the three variables `$``, `$&` or `$'` (or their `use English` equivalents) anywhere in the code, caused all subsequent successful pattern matches to make a copy of the matched string, in case the code might subsequently access one of those variables. This imposed a considerable performance penalty across the whole program, so generally the use of these variables has been discouraged.

In Perl 5.6.0 the `@-` and `@+` dynamic arrays were introduced that supply the indices of successful matches. So you could for example do this:

```
$str =~ /pattern/;

print $`, $&, $'; # bad: perfomance hit

print             # good: no perfomance hit
    substr($str, 0,      $-[0]),
    substr($str, $-[0], $+[0]-$-[0]),
    substr($str, $+[0]);
```

In Perl 5.10.0 the `/p` match operator flag and the `${^PREMATCH}`, `${^MATCH}`, and `${^POSTMATCH}` variables were introduced, that allowed you to suffer the penalties only on patterns marked with `/p`.

In Perl 5.18.0 onwards, perl started noting the presence of each of the three variables separately, and only copied that part of the string required; so in

```
$`; $&; "abcdefgh" =~ /d/
```

perl would only copy the ‘‘abcd’’ part of the string. That could make a big difference in something like

```
$str = 'x' x 1_000_000;
$&; # whoops
$str =~ /x/g # one char copied a million times, not a million chars
```

In Perl 5.20.0 a new copy-on-write system was enabled by default, which finally fixes all performance issues with these three variables, and makes them safe to use anywhere.

The `Devel::NYTProf` and `Devel::FindAmpersand` modules can help you find uses of these problematic match variables in your code.

$<*digits*> ($1, `$2`, ...)
: Contains the subpattern from the corresponding set of capturing parentheses from the last successful pattern match, not counting patterns matched in nested blocks that have been exited already.

  Note there is a distinction between a capture buffer which matches the empty string a capture buffer which is optional. Eg, `(x?)` and `(x)?` The latter may be undef, the former not.

  These variables are read-only and dynamically-scoped.

  Mnemonic: like \digits.

@{^CAPTURE}
: An array which exposes the contents of the capture buffers, if any, of the last successful pattern match, not counting patterns matched in nested blocks that have been exited already.

Note that the 0 index of @{^CAPTURE} is equivalent to $1, the 1 index is equivalent to $2, etc.

```
if ("foal"=~/(.)(.)(.)(.)/) {
    print join "-", @{^CAPTURE};
}
```

should output "f–o-a-l".

See also "$*digits*", "*%*{^CAPTURE}" and "*%*{^CAPTURE_ALL}".

Note that unlike most other regex magic variables there is no single letter equivalent to @{^CAPTURE}.

This variable was added in 5.25.7

$MATCH

$& The string matched by the last successful pattern match (not counting any matches hidden within a BLOCK or eval() enclosed by the current BLOCK).

See "Performance issues" above for the serious performance implications of using this variable (even once) in your code.

This variable is read-only and dynamically-scoped.

Mnemonic: like & in some editors.

${^MATCH}

This is similar to $& ($MATCH) except that it does not incur the performance penalty associated with that variable.

See "Performance issues" above.

In Perl v5.18 and earlier, it is only guaranteed to return a defined value when the pattern was compiled or executed with the /p modifier. In Perl v5.20, the /p modifier does nothing, so ${^MATCH} does the same thing as $MATCH.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

$PREMATCH

$‘ The string preceding whatever was matched by the last successful pattern match, not counting any matches hidden within a BLOCK or eval enclosed by the current BLOCK.

See "Performance issues" above for the serious performance implications of using this variable (even once) in your code.

This variable is read-only and dynamically-scoped.

Mnemonic: ` often precedes a quoted string.

${^PREMATCH}

This is similar to $` ($PREMATCH) except that it does not incur the performance penalty associated with that variable.

See "Performance issues" above.

In Perl v5.18 and earlier, it is only guaranteed to return a defined value when the pattern was compiled or executed with the /p modifier. In Perl v5.20, the /p modifier does nothing, so ${^PREMATCH} does the same thing as $PREMATCH.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

$POSTMATCH

$’ The string following whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or eval() enclosed by the current BLOCK). Example:

```
local $_ = 'abcdefghi';
/def/;
print "$`:$&:$'\n";          # prints abc:def:ghi
```

See "Performance issues" above for the serious performance implications of using this variable (even once) in your code.

This variable is read-only and dynamically-scoped.

Mnemonic: ' often follows a quoted string.

${^POSTMATCH}

This is similar to $' ($POSTMATCH) except that it does not incur the performance penalty associated with that variable.

See "Performance issues" above.

In Perl v5.18 and earlier, it is only guaranteed to return a defined value when the pattern was compiled or executed with the /p modifier. In Perl v5.20, the /p modifier does nothing, so ${^POSTMATCH} does the same thing as $POSTMATCH.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

$LAST_PAREN_MATCH

$+ The text matched by the last bracket of the last successful search pattern. This is useful if you don't know which one of a set of alternative patterns matched. For example:

```
/Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

This variable is read-only and dynamically-scoped.

Mnemonic: be positive and forward looking.

$LAST_SUBMATCH_RESULT

$^N The text matched by the used group most-recently closed (i.e. the group with the rightmost closing parenthesis) of the last successful search pattern.

This is primarily used inside (?{...}) blocks for examining text recently matched. For example, to effectively capture text to a variable (in addition to $1, $2, etc.), replace (...) with

```
(?:(...)(?{ $var = $^N }))
```

By setting and then using $var in this way relieves you from having to worry about exactly which numbered set of parentheses they are.

This variable was added in Perl v5.8.0.

Mnemonic: the (possibly) Nested parenthesis that most recently closed.

@LAST_MATCH_END

@+ This array holds the offsets of the ends of the last successful submatches in the currently active dynamic scope. $+[0] is the offset into the string of the end of the entire match. This is the same value as what the pos function returns when called on the variable that was matched against. The *n*th element of this array holds the offset of the *n*th submatch, so $+[1] is the offset past where $1 ends, $+[2] the offset past where $2 ends, and so on. You can use $#+ to determine how many subgroups were in the last successful match. See the examples given for the @- variable.

This variable was added in Perl v5.6.0.

%{^CAPTURE}

%LAST_PAREN_MATCH

%+ Similar to @+, the %+ hash allows access to the named capture buffers, should they exist, in the last successful match in the currently active dynamic scope.

For example, $+{foo} is equivalent to $1 after the following match:

```
'foo' =~ /(?<foo>foo)/;
```

The keys of the `%+` hash list only the names of buffers that have captured (and that are thus associated to defined values).

The underlying behaviour of `%+` is provided by the Tie::Hash::NamedCapture module.

Note: `%-` and `%+` are tied views into a common internal hash associated with the last successful regular expression. Therefore mixing iterative access to them via `each` may have unpredictable results. Likewise, if the last successful match changes, then the results may be surprising.

This variable was added in Perl v5.10.0. The `%{^CAPTURE}` alias was added in 5.25.7.

This variable is read-only and dynamically-scoped.

`@LAST_MATCH_START`

`@-` `$-[0]` is the offset of the start of the last successful match. `$-[`*n*`]` is the offset of the start of the substring matched by *n*–th subpattern, or undef if the subpattern did not match.

Thus, after a match against `$_`, `$&` coincides with `substr $_, $-[0], $+[0] - $-[0]`. Similarly, $*n* coincides with `substr $_, $-[n], $+[n] - $-[n]` if `$-[n]` is defined, and $+ coincides with `substr $_, $-[$#-], $+[$#-] - $-[$#-]`. One can use `$#-` to find the last matched subgroup in the last successful match. Contrast with `$#+`, the number of subgroups in the regular expression. Compare with `@+`.

This array holds the offsets of the beginnings of the last successful submatches in the currently active dynamic scope. `$-[0]` is the offset into the string of the beginning of the entire match. The *n*th element of this array holds the offset of the *n*th submatch, so `$-[1]` is the offset where `$1` begins, `$-[2]` the offset where `$2` begins, and so on.

After a match against some variable `$var`:

`$`` is the same as `substr($var, 0, $-[0])`
`$&` is the same as `substr($var, $-[0], $+[0] - $-[0])`
`$'` is the same as `substr($var, $+[0])`
`$1` is the same as `substr($var, $-[1], $+[1] - $-[1])`
`$2` is the same as `substr($var, $-[2], $+[2] - $-[2])`
`$3` is the same as `substr($var, $-[3], $+[3] - $-[3])`

This variable was added in Perl v5.6.0.

%{^CAPTURE_ALL}

`%-` Similar to `%+`, this variable allows access to the named capture groups in the last successful match in the currently active dynamic scope. To each capture group name found in the regular expression, it associates a reference to an array containing the list of values captured by all buffers with that name (should there be several of them), in the order where they appear.

Here's an example:

```
if ('1234' =~ /(?<A>1)(?<B>2)(?<A>3)(?<B>4)/) {
    foreach my $bufname (sort keys %-) {
        my $ary = $-{$bufname};
        foreach my $idx (0..$#$ary) {
            print "\$-{$bufname}[$idx] : ",
                  (defined($ary->[$idx])
                      ? "'$ary->[$idx]'"
                      : "undef"),
                  "\n";
        }
    }
}
```

would print out:

```
$-{A}[0] : '1'
$-{A}[1] : '3'
$-{B}[0] : '2'
$-{B}[1] : '4'
```

The keys of the `%-` hash correspond to all buffer names found in the regular expression.

The behaviour of `%-` is implemented via the Tie::Hash::NamedCapture module.

Note: `%-` and `%+` are tied views into a common internal hash associated with the last successful regular expression. Therefore mixing iterative access to them via `each` may have unpredictable results. Likewise, if the last successful match changes, then the results may be surprising.

This variable was added in Perl v5.10.0. The `%{^CAPTURE_ALL}` alias was added in 5.25.7.

This variable is read-only and dynamically-scoped.

`$LAST_REGEXP_CODE_RESULT`

$^R The result of evaluation of the last successful `(?{ code })` regular expression assertion (see perlre). May be written to.

This variable was added in Perl 5.005.

${^RE_DEBUG_FLAGS}

The current value of the regex debugging flags. Set to 0 for no debug output even when the `re 'debug'` module is loaded. See re for details.

This variable was added in Perl v5.10.0.

${^RE_TRIE_MAXBUF}

Controls how certain regex optimisations are applied and how much memory they utilize. This value by default is 65536 which corresponds to a 512kB temporary cache. Set this to a higher value to trade memory for speed when matching large alternations. Set it to a lower value if you want the optimisations to be as conservative of memory as possible but still occur, and set it to a negative value to prevent the optimisation and conserve the most memory. Under normal situations this variable should be of no interest to you.

This variable was added in Perl v5.10.0.

Variables related to filehandles

Variables that depend on the currently selected filehandle may be set by calling an appropriate object method on the `IO::Handle` object, although this is less efficient than using the regular built-in variables. (Summary lines below for this contain the word HANDLE.) First you must say

```
use IO::Handle;
```

after which you may use either

```
method HANDLE EXPR
```

or more safely,

```
HANDLE->method(EXPR)
```

Each method returns the old value of the `IO::Handle` attribute. The methods each take an optional EXPR, which, if supplied, specifies the new value for the `IO::Handle` attribute in question. If not supplied, most methods do nothing to the current value — except for `autoflush()`, which will assume a 1 for you, just to be different.

Because loading in the `IO::Handle` class is an expensive operation, you should learn how to use the regular built-in variables.

A few of these variables are considered "read-only". This means that if you try to assign to this variable, either directly or indirectly through a reference, you'll raise a run-time exception.

You should be very careful when modifying the default values of most special variables described in this document. In most cases you want to localize these variables before changing them, since if you don't, the change may affect other modules which rely on the default values of the special variables that you have changed. This is one of the correct ways to read the whole file at once:

```
open my $fh, "<", "foo" or die $!;
local $/; # enable localized slurp mode
my $content = <$fh>;
close $fh;
```

But the following code is quite bad:

```
open my $fh, "<", "foo" or die $!;
undef $/; # enable slurp mode
my $content = <$fh>;
close $fh;
```

since some other module, may want to read data from some file in the default ‘‘line mode’’, so if the code we have just presented has been executed, the global value of `$/` is now changed for any other code running inside the same Perl interpreter.

Usually when a variable is localized you want to make sure that this change affects the shortest scope possible. So unless you are already inside some short `{}` block, you should create one yourself. For example:

```
my $content = '';
open my $fh, "<", "foo" or die $!;
{
    local $/;
    $content = <$fh>;
}
close $fh;
```

Here is an example of how your own code can go broken:

```
for ( 1..3 ){
    $\ = "\r\n";
    nasty_break();
    print "$_";
}

sub nasty_break {
    $\ = "\f";
    # do something with $_
}
```

You probably expect this code to print the equivalent of

```
"1\r\n2\r\n3\r\n"
```

but instead you get:

```
"1\f2\f3\f"
```

Why? Because `nasty_break()` modifies `$\` without localizing it first. The value you set in `nasty_break()` is still there when you return. The fix is to add `local()` so the value doesn't leak out of `nasty_break()`:

```
local $\ = "\f";
```

It's easy to notice the problem in such a short example, but in more complicated code you are looking for trouble if you don't localize changes to the special variables.

`$ARGV` Contains the name of the current file when reading from <>.

`@ARGV` The array `@ARGV` contains the command-line arguments intended for the script. `$#ARGV` is generally the number of arguments minus one, because `$ARGV[0]` is the first argument, *not* the program's command name itself. See ‘‘$0’’ for the command name.

ARGV The special filehandle that iterates over command-line filenames in `@ARGV`. Usually written as the null filehandle in the angle operator <>. Note that currently `ARGV` only has its magical effect within the <> operator; elsewhere it is just a plain filehandle corresponding to the last file opened by <>. In particular, passing `\*ARGV` as a parameter to a function that expects a filehandle may not cause your function to automatically read the contents of all the files in `@ARGV`.

ARGVOUT

The special filehandle that points to the currently open output file when doing edit-in-place processing with **–i**. Useful when you have to do a lot of inserting and don't want to keep modifying `$_`. See perlrun for the **–i** switch.

IO::Handle–>output_field_separator(EXPR)
`$OUTPUT_FIELD_SEPARATOR`
`$OFS`
$,

The output field separator for the print operator. If defined, this value is printed between each of print's arguments. Default is `undef`.

You cannot call `output_field_separator()` on a handle, only as a static method. See IO::Handle.

Mnemonic: what is printed when there is a "," in your print statement.

HANDLE–>input_line_number(EXPR)
`$INPUT_LINE_NUMBER`
`$NR`
$.

Current line number for the last filehandle accessed.

Each filehandle in Perl counts the number of lines that have been read from it. (Depending on the value of `$/`, Perl's idea of what constitutes a line may not match yours.) When a line is read from a filehandle (via `readline()` or <>), or when `tell()` or `seek()` is called on it, `$.` becomes an alias to the line counter for that filehandle.

You can adjust the counter by assigning to `$.`, but this will not actually move the seek pointer. *Localizing `$.` will not localize the filehandle's line count*. Instead, it will localize perl's notion of which filehandle `$.` is currently aliased to.

`$.` is reset when the filehandle is closed, but **not** when an open filehandle is reopened without an intervening `close()`. For more details, see "I/O Operators" in perlop. Because <> never does an explicit close, line numbers increase across `ARGV` files (but see examples in "eof" in perlfunc).

You can also use `HANDLE->input_line_number(EXPR)` to access the line counter for a given filehandle without having to worry about which handle you last accessed.

Mnemonic: many programs use "." to mean the current line number.

IO::Handle–>input_record_separator(EXPR)
`$INPUT_RECORD_SEPARATOR`
`$RS`
$/

The input record separator, newline by default. This influences Perl's idea of what a "line" is. Works like **awk**'s RS variable, including treating empty lines as a terminator if set to the null string (an empty line cannot contain any spaces or tabs). You may set it to a multi-character string to match a multi-character terminator, or to `undef` to read through the end of file. Setting it to `"\n\n"` means something slightly different than setting to `""`, if the file contains consecutive empty lines. Setting to `""` will treat two or more consecutive empty lines as a single empty line. Setting to `"\n\n"` will blindly assume that the next input character belongs to the next paragraph, even if it's a newline.

```
local $/;           # enable "slurp" mode
local $_ = <FH>;    # whole file now here
s/\n[ \t]+/ /g;
```

Remember: the value of `$/` is a string, not a regex. **awk** has to be better for something. :–)

Setting `$/` to a reference to an integer, scalar containing an integer, or scalar that's convertible to an integer will attempt to read records instead of lines, with the maximum record size being the referenced integer number of characters. So this:

```
local $/ = \32768; # or \"32768", or \$var_containing_32768
open my $fh, "<", $myfile or die $!;
local $_ = <$fh>;
```

will read a record of no more than 32768 characters from `$fh`. If you're not reading from a record-oriented file (or your OS doesn't have record-oriented files), then you'll likely get a full chunk of data with every read. If a record is larger than the record size you've set, you'll get the record back in pieces. Trying to set the record size to zero or less is deprecated and will cause $/ to have the value of "undef", which will cause reading in the (rest of the) whole file.

As of 5.19.9 setting `$/` to any other form of reference will throw a fatal exception. This is in preparation for supporting new ways to set `$/` in the future.

On VMS only, record reads bypass PerlIO layers and any associated buffering, so you must not mix record and non-record reads on the same filehandle. Record mode mixes with line mode only when the same buffering layer is in use for both modes.

You cannot call `input_record_separator()` on a handle, only as a static method. See IO::Handle.

See also "Newlines" in perlport. Also see "$.".

Mnemonic: / delimits line boundaries when quoting poetry.

IO::Handle->output_record_separator(EXPR)
`$OUTPUT_RECORD_SEPARATOR`
`$ORS`

$\ The output record separator for the print operator. If defined, this value is printed after the last of print's arguments. Default is `undef`.

You cannot call `output_record_separator()` on a handle, only as a static method. See IO::Handle.

Mnemonic: you set `$\` instead of adding "\n" at the end of the print. Also, it's just like `$/`, but it's what you get "back" from Perl.

HANDLE->autoflush(EXPR)
`$OUTPUT_AUTOFLUSH`

$| If set to nonzero, forces a flush right away and after every write or print on the currently selected output channel. Default is 0 (regardless of whether the channel is really buffered by the system or not; `$|` tells you only whether you've asked Perl explicitly to flush after each write). STDOUT will typically be line buffered if output is to the terminal and block buffered otherwise. Setting this variable is useful primarily when you are outputting to a pipe or socket, such as when you are running a Perl program under **rsh** and want to see the output as it's happening. This has no effect on input buffering. See "getc" in perlfunc for that. See "select" in perlfunc on how to select the output channel. See also IO::Handle.

Mnemonic: when you want your pipes to be piping hot.

${^LAST_FH}

This read-only variable contains a reference to the last-read filehandle. This is set by `<HANDLE>`, `readline`, `tell`, `eof` and `seek`. This is the same handle that `$.` and `tell` and `eof` without arguments use. It is also the handle used when Perl appends ", <STDIN> line 1" to an error or warning message.

This variable was added in Perl v5.18.0.

Variables related to formats

The special variables for formats are a subset of those for filehandles. See perlform for more information about Perl's formats.

`$ACCUMULATOR`

$^A The current value of the `write()` accumulator for `format()` lines. A format contains `formline()` calls that put their result into `$^A`. After calling its format, `write()` prints out the contents of `$^A` and empties. So you never really see the contents of `$^A` unless you call `formline()` yourself and then look at it. See perlform and "formline PICTURE,LIST" in perlfunc.

IO::Handle->format_formfeed(EXPR)
`$FORMAT_FORMFEED`
$^L What formats output as a form feed. The default is `\f`.

You cannot call `format_formfeed()` on a handle, only as a static method. See IO::Handle.

HANDLE->format_page_number(EXPR)
`$FORMAT_PAGE_NUMBER`
$% The current page number of the currently selected output channel.

Mnemonic: `%` is page number in **nroff**.

HANDLE->format_lines_left(EXPR)
`$FORMAT_LINES_LEFT`
$- The number of lines left on the page of the currently selected output channel.

Mnemonic: lines_on_page - lines_printed.

IO::Handle->format_line_break_characters EXPR
`$FORMAT_LINE_BREAK_CHARACTERS`
`$:` The current set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format. The default is " \n-", to break on a space, newline, or a hyphen.

You cannot call `format_line_break_characters()` on a handle, only as a static method. See IO::Handle.

Mnemonic: a "colon" in poetry is a part of a line.

HANDLE->format_lines_per_page(EXPR)
`$FORMAT_LINES_PER_PAGE`
$= The current page length (printable lines) of the currently selected output channel. The default is 60.

Mnemonic: = has horizontal lines.

HANDLE->format_top_name(EXPR)
`$FORMAT_TOP_NAME`
$^ The name of the current top-of-page format for the currently selected output channel. The default is the name of the filehandle with `_TOP` appended. For example, the default format top name for the `STDOUT` filehandle is `STDOUT_TOP`.

Mnemonic: points to top of page.

HANDLE->format_name(EXPR)
`$FORMAT_NAME`
$~ The name of the current report format for the currently selected output channel. The default format name is the same as the filehandle name. For example, the default format name for the `STDOUT` filehandle is just `STDOUT`.

Mnemonic: brother to `$^`.

Error Variables

The variables `$@`, `$!`, `$^E`, and `$?` contain information about different types of error conditions that may appear during execution of a Perl program. The variables are shown ordered by the "distance" between the subsystem which reported the error and the Perl process. They correspond to errors detected by the Perl interpreter, C library, operating system, or an external program, respectively.

To illustrate the differences between these variables, consider the following Perl expression, which uses a single-quoted string. After execution of this statement, perl may have set all four special error variables:

```
eval q{
    open my $pipe, "/cdrom/install |" or die $!;
    my @res = <$pipe>;
    close $pipe or die "bad pipe: $?, $!";
};
```

When perl executes the `eval()` expression, it translates the `open()`, `<PIPE>`, and `close` calls in the C run-time library and thence to the operating system kernel. perl sets `$!` to the C library's `errno` if one of these calls fails.

$@ is set if the string to be eval−ed did not compile (this may happen if open or close were imported with bad prototypes), or if Perl code executed during evaluation die()d. In these cases the value of $@ is the compile error, or the argument to die (which will interpolate $! and $?). (See also Fatal, though.)

Under a few operating systems, $^E may contain a more verbose error indicator, such as in this case, "CDROM tray not closed." Systems that do not support extended error messages leave $^E the same as $!.

Finally, $? may be set to a non−0 value if the external program */cdrom/install* fails. The upper eight bits reflect specific error conditions encountered by the program (the program's exit() value). The lower eight bits reflect mode of failure, like signal death and core dump information. See *wait*(2) for details. In contrast to $! and $^E, which are set only if an error condition is detected, the variable $? is set on each wait or pipe close, overwriting the old value. This is more like $@, which on every eval() is always set on failure and cleared on success.

For more details, see the individual descriptions at $@, $!, $^E, and $?.

${^CHILD_ERROR_NATIVE}

The native status returned by the last pipe close, backtick (``) command, successful call to wait() or waitpid(), or from the system() operator. On POSIX-like systems this value can be decoded with the WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG and WIFCONTINUED functions provided by the POSIX module.

Under VMS this reflects the actual VMS exit status; i.e. it is the same as $? when the pragma use vmsish 'status' is in effect.

This variable was added in Perl v5.10.0.

$EXTENDED_OS_ERROR

$^E Error information specific to the current operating system. At the moment, this differs from "$!" under only VMS, OS/2, and Win32 (and for MacPerl). On all other platforms, $^E is always just the same as $!.

Under VMS, $^E provides the VMS status value from the last system error. This is more specific information about the last system error than that provided by $!. This is particularly important when $! is set to **EVMSERR**.

Under OS/2, $^E is set to the error code of the last call to OS/2 API either via CRT, or directly from perl.

Under Win32, $^E always returns the last error information reported by the Win32 call GetLastError() which describes the last error from within the Win32 API. Most Win32−specific code will report errors via $^E. ANSI C and Unix-like calls set errno and so most portable Perl code will report errors via $!.

Caveats mentioned in the description of "$!" generally apply to $^E, also.

This variable was added in Perl 5.003.

Mnemonic: Extra error explanation.

$EXCEPTIONS_BEING_CAUGHT

$^S Current state of the interpreter.

```
$^S         State
---------   -------------------------------------
undef       Parsing module, eval, or main program
true (1)    Executing an eval
false (0)   Otherwise
```

The first state may happen in $SIG{__DIE__} and $SIG{__WARN__} handlers.

The English name $EXCEPTIONS_BEING_CAUGHT is slightly misleading, because the undef value does not indicate whether exceptions are being caught, since compilation of the main program does not catch exceptions.

This variable was added in Perl 5.004.

`$WARNING`
$^W The current value of the warning switch, initially true if **-w** was used, false otherwise, but directly modifiable.

See also warnings.

Mnemonic: related to the **-w** switch.

${^WARNING_BITS}
The current set of warning checks enabled by the `use warnings` pragma. It has the same scoping as the `$^H` and `%^H` variables. The exact values are considered internal to the warnings pragma and may change between versions of Perl.

This variable was added in Perl v5.6.0.

`$OS_ERROR`
`$ERRNO`
$! When referenced, `$!` retrieves the current value of the C `errno` integer variable. If `$!` is assigned a numerical value, that value is stored in `errno`. When referenced as a string, `$!` yields the system error string corresponding to `errno`.

Many system or library calls set `errno` if they fail, to indicate the cause of failure. They usually do **not** set `errno` to zero if they succeed. This means `errno`, hence `$!`, is meaningful only *immediately* after a **failure**:

```
if (open my $fh, "<", $filename) {
            # Here $! is meaningless.
            ...
}
else {
            # ONLY here is $! meaningful.
            ...
            # Already here $! might be meaningless.
}
# Since here we might have either success or failure,
# $! is meaningless.
```

Here, *meaningless* means that `$!` may be unrelated to the outcome of the `open()` operator. Assignment to `$!` is similarly ephemeral. It can be used immediately before invoking the `die()` operator, to set the exit value, or to inspect the system error string corresponding to error *n*, or to restore `$!` to a meaningful state.

Mnemonic: What just went bang?

`%OS_ERROR`
`%ERRNO`
%! Each element of `%!` has a true value only if `$!` is set to that value. For example, `$!{ENOENT}` is true if and only if the current value of `$!` is `ENOENT`; that is, if the most recent error was "No such file or directory" (or its moral equivalent: not all operating systems give that exact error, and certainly not all languages). The specific true value is not guaranteed, but in the past has generally been the numeric value of `$!`. To check if a particular key is meaningful on your system, use `exists $!{the_key}`; for a list of legal keys, use `keys %!`. See Errno for more information, and also see "$!".

This variable was added in Perl 5.005.

`$CHILD_ERROR`
$? The status returned by the last pipe close, backtick (``) command, successful call to `wait()` or `waitpid()`, or from the `system()` operator. This is just the 16-bit status word returned by the traditional Unix `wait()` system call (or else is made up to look like it). Thus, the exit value of the subprocess is really (`$? >> 8`), and `$? & 127` gives which signal, if any, the process died from, and `$? & 128` reports whether there was a core dump.

Additionally, if the `h_errno` variable is supported in C, its value is returned via `$?` if any `gethost*()` function fails.

If you have installed a signal handler for SIGCHLD, the value of $? will usually be wrong outside that handler.

Inside an END subroutine $? contains the value that is going to be given to exit(). You can modify $? in an END subroutine to change the exit status of your program. For example:

```
END {
    $? = 1 if $? == 255;  # die would make it 255
}
```

Under VMS, the pragma use vmsish 'status' makes $? reflect the actual VMS exit status, instead of the default emulation of POSIX status; see "$?" in perlvms for details.

Mnemonic: similar to **sh** and **ksh**.

$EVAL_ERROR

$@ The Perl error from the last eval operator, i.e. the last exception that was caught. For eval BLOCK, this is either a runtime error message or the string or reference die was called with. The eval STRING form also catches syntax errors and other compile time exceptions.

If no error occurs, eval sets $@ to the empty string.

Warning messages are not collected in this variable. You can, however, set up a routine to process warnings by setting $SIG{__WARN__} as described in "%SIG".

Mnemonic: Where was the error "at"?

Variables related to the interpreter state

These variables provide information about the current interpreter state.

$COMPILING

$^C The current value of the flag associated with the **-c** switch. Mainly of use with **-MO=...** to allow code to alter its behavior when being compiled, such as for example to AUTOLOAD at compile time rather than normal, deferred loading. Setting $^C = 1 is similar to calling B::minus_c.

This variable was added in Perl v5.6.0.

$DEBUGGING

$^D The current value of the debugging flags. May be read or set. Like its command-line equivalent, you can use numeric or symbolic values, e.g. $^D = 10 or $^D = "st". See "**-D***number*" in perlrun. The contents of this variable also affects the debugger operation. See "Debugger Internals" in perldebguts.

Mnemonic: value of **-D** switch.

${^ENCODING}

This variable is no longer supported.

It used to hold the *object reference* to the Encode object that was used to convert the source code to Unicode.

Its purpose was to allow your non-ASCII Perl scripts not to have to be written in UTF-8; this was useful before editors that worked on UTF-8 encoded text were common, but that was long ago. It caused problems, such as affecting the operation of other modules that weren't expecting it, causing general mayhem.

If you need something like this functionality, it is recommended that use you a simple source filter, such as Filter::Encoding.

If you are coming here because code of yours is being adversely affected by someone's use of this variable, you can usually work around it by doing this:

```
local ${^ENCODING};
```

near the beginning of the functions that are getting broken. This undefines the variable during the scope of execution of the including function.

This variable was added in Perl 5.8.2 and removed in 5.26.0.

${^GLOBAL_PHASE}

The current phase of the perl interpreter.

Possible values are:

CONSTRUCT

The `PerlInterpreter*` is being constructed via `perl_construct`. This value is mostly there for completeness and for use via the underlying C variable `PL_phase`. It's not really possible for Perl code to be executed unless construction of the interpreter is finished.

START

This is the global compile-time. That includes, basically, every `BEGIN` block executed directly or indirectly from during the compile-time of the top-level program.

This phase is not called "BEGIN" to avoid confusion with `BEGIN`-blocks, as those are executed during compile-time of any compilation unit, not just the top-level program. A new, localised compile-time entered at run-time, for example by constructs as `eval "use SomeModule"` are not global interpreter phases, and therefore aren't reflected by `${^GLOBAL_PHASE}`.

CHECK

Execution of any `CHECK` blocks.

INIT

Similar to "CHECK", but for `INIT`-blocks, not `CHECK` blocks.

RUN

The main run-time, i.e. the execution of `PL_main_root`.

END

Execution of any `END` blocks.

DESTRUCT

Global destruction.

Also note that there's no value for UNITCHECK-blocks. That's because those are run for each compilation unit individually, and therefore is not a global interpreter phase.

Not every program has to go through each of the possible phases, but transition from one phase to another can only happen in the order described in the above list.

An example of all of the phases Perl code can see:

```
BEGIN { print "compile-time: ${^GLOBAL_PHASE}\n" }

INIT  { print "init-time: ${^GLOBAL_PHASE}\n" }

CHECK { print "check-time: ${^GLOBAL_PHASE}\n" }

{
    package Print::Phase;

    sub new {
        my ($class, $time) = @_;
        return bless \$time, $class;
    }

    sub DESTROY {
        my $self = shift;
        print "$$self: ${^GLOBAL_PHASE}\n";
    }
}

print "run-time: ${^GLOBAL_PHASE}\n";

my $runtime = Print::Phase->new(
    "lexical variables are garbage collected before END"
);

END   { print "end-time: ${^GLOBAL_PHASE}\n" }
```

```
our $destruct = Print::Phase->new(
    "package variables are garbage collected after END"
);
```

This will print out

```
compile-time: START
check-time: CHECK
init-time: INIT
run-time: RUN
lexical variables are garbage collected before END: RUN
end-time: END
package variables are garbage collected after END: DESTRUCT
```

This variable was added in Perl 5.14.0.

$^H WARNING: This variable is strictly for internal use only. Its availability, behavior, and contents are subject to change without notice.

This variable contains compile-time hints for the Perl interpreter. At the end of compilation of a BLOCK the value of this variable is restored to the value when the interpreter started to compile the BLOCK.

When perl begins to parse any block construct that provides a lexical scope (e.g., eval body, required file, subroutine body, loop body, or conditional block), the existing value of `$^H` is saved, but its value is left unchanged. When the compilation of the block is completed, it regains the saved value. Between the points where its value is saved and restored, code that executes within BEGIN blocks is free to change the value of `$^H`.

This behavior provides the semantic of lexical scoping, and is used in, for instance, the `use strict` pragma.

The contents should be an integer; different bits of it are used for different pragmatic flags. Here's an example:

```
sub add_100 { $^H |= 0x100 }

sub foo {
    BEGIN { add_100() }
    bar->baz($boon);
}
```

Consider what happens during execution of the BEGIN block. At this point the BEGIN block has already been compiled, but the body of `foo()` is still being compiled. The new value of `$^H` will therefore be visible only while the body of `foo()` is being compiled.

Substitution of `BEGIN { add_100() }` block with:

```
BEGIN { require strict; strict->import('vars') }
```

demonstrates how `use strict 'vars'` is implemented. Here's a conditional version of the same lexical pragma:

```
BEGIN {
    require strict; strict->import('vars') if $condition
}
```

This variable was added in Perl 5.003.

%^H The `%^H` hash provides the same scoping semantic as `$^H`. This makes it useful for implementation of lexically scoped pragmas. See perlpragma. All the entries are stringified when accessed at runtime, so only simple values can be accommodated. This means no pointers to objects, for example.

When putting items into `%^H`, in order to avoid conflicting with other users of the hash there is a convention regarding which keys to use. A module should use only keys that begin with the

module's name (the name of its main package) and a "/" character. For example, a module `Foo::Bar` should use keys such as `Foo::Bar/baz`.

This variable was added in Perl v5.6.0.

${^OPEN}
: An internal variable used by PerlIO. A string in two parts, separated by a `\0` byte, the first part describes the input layers, the second part describes the output layers.

  This variable was added in Perl v5.8.0.

`$PERLDB`
$^P
: The internal variable for debugging support. The meanings of the various bits are subject to change, but currently indicate:

  0x01 Debug subroutine enter/exit.

  0x02 Line-by-line debugging. Causes `DB::DB()` subroutine to be called for each statement executed. Also causes saving source code lines (like 0x400).

  0x04 Switch off optimizations.

  0x08 Preserve more data for future interactive inspections.

  0x10 Keep info about source lines on which a subroutine is defined.

  0x20 Start with single-step on.

  0x40 Use subroutine address instead of name when reporting.

  0x80 Report `goto &subroutine` as well.

  0x100 Provide informative "file" names for evals based on the place they were compiled.

  0x200 Provide informative names to anonymous subroutines based on the place they were compiled.

  0x400 Save source code lines into `@{"_<$filename"}`.

  0x800 When saving source, include evals that generate no subroutines.

  0x1000
  : When saving source, include source that did not compile.

  Some bits may be relevant at compile-time only, some at run-time only. This is a new mechanism and the details may change. See also perldebguts.

${^TAINT}
: Reflects if taint mode is on or off. 1 for on (the program was run with **-T**), 0 for off, -1 when only taint warnings are enabled (i.e. with **-t** or **-TU**).

  This variable is read-only.

  This variable was added in Perl v5.8.0.

${^UNICODE}
: Reflects certain Unicode settings of Perl. See perlrun documentation for the `-C` switch for more information about the possible values.

  This variable is set during Perl startup and is thereafter read-only.

  This variable was added in Perl v5.8.2.

${^UTF8CACHE}
: This variable controls the state of the internal UTF-8 offset caching code. 1 for on (the default), 0 for off, -1 to debug the caching code by checking all its results against linear scans, and panicking on any discrepancy.

  This variable was added in Perl v5.8.9. It is subject to change or removal without notice, but is currently used to avoid recalculating the boundaries of multi-byte UTF-8-encoded characters.

${^UTF8LOCALE}
: This variable indicates whether a UTF-8 locale was detected by perl at startup. This information is used by perl when it's in adjust-utf8ness-to-locale mode (as when run with the `-CL`

command-line switch); see perlrun for more info on this.

This variable was added in Perl v5.8.8.

Deprecated and removed variables

Deprecating a variable announces the intent of the perl maintainers to eventually remove the variable from the language. It may still be available despite its status. Using a deprecated variable triggers a warning.

Once a variable is removed, its use triggers an error telling you the variable is unsupported.

See perldiag for details about error messages.

$# `$#` was a variable that could be used to format printed numbers. After a deprecation cycle, its magic was removed in Perl v5.10.0 and using it now triggers a warning: `$# is no longer supported`.

This is not the sigil you use in front of an array name to get the last index, like `$#array`. That's still how you get the last index of an array in Perl. The two have nothing to do with each other.

Deprecated in Perl 5.

Removed in Perl v5.10.0.

$* `$*` was a variable that you could use to enable multiline matching. After a deprecation cycle, its magic was removed in Perl v5.10.0. Using it now triggers a warning: `$* is no longer supported`. You should use the `/s` and `/m` regexp modifiers instead.

Deprecated in Perl 5.

Removed in Perl v5.10.0.

$[This variable stores the index of the first element in an array, and of the first character in a substring. The default is 0, but you could theoretically set it to 1 to make Perl behave more like **awk** (or Fortran) when subscripting and when evaluating the *index()* and *substr()* functions.

As of release 5 of Perl, assignment to `$[` is treated as a compiler directive, and cannot influence the behavior of any other file. (That's why you can only assign compile-time constants to it.) Its use is highly discouraged.

Prior to Perl v5.10.0, assignment to `$[` could be seen from outer lexical scopes in the same file, unlike other compile-time directives (such as strict). Using *local()* on it would bind its value strictly to a lexical block. Now it is always lexically scoped.

As of Perl v5.16.0, it is implemented by the arybase module. See arybase for more details on its behaviour.

Under `use v5.16`, or `no feature "array_base"`, `$[` no longer has any effect, and always contains 0. Assigning 0 to it is permitted, but any other value will produce an error.

Mnemonic: [begins subscripts.

Deprecated in Perl v5.12.0.

NAME

perlrun – how to execute the Perl interpreter

SYNOPSIS

perl [**–sTtuUWX**] [**–hv**] [**–V**[:*configvar*]]
[**–cw**] [**–d**[**t**][:*debugger*]] [**–D**[*number/list*]]
[**–pna**] [**–F***pattern*] [**–l**[*octal*]] [**–0**[*octal/hexadecimal*]]
[**–I***dir*] [**–m**[**–**]*module*] [**–M**[**–**]*'module...'*] [**–f**] [**–C [*number/list*]**] [**–S**]
[**–x**[*dir*]] [**–i**[*extension*]]
[[**–e**|**–E**] *'command'*] [**––**] [*programfile*] [*argument*]...

DESCRIPTION

The normal way to run a Perl program is by making it directly executable, or else by passing the name of the source file as an argument on the command line. (An interactive Perl environment is also possible — see perldebug for details on how to do that.) Upon startup, Perl looks for your program in one of the following places:

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

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

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

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

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

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

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

to let Perl see the **–p** switch.

A similar trick involves the *env* program, if you have it.

```
#!/usr/bin/env perl
```

The examples above use a relative path to the perl interpreter, getting whatever version is first in the user's path. If you want a specific version of Perl, say, perl5.14.1, you should place that directly in the `#!` line's path.

If the `#!` line does not contain the word ''perl'' nor the word ''indir'', the program named after the `#!` is executed instead of the Perl interpreter. This is slightly bizarre, but it helps people on machines that don't do `#!`, because they can tell a program that their SHELL is */usr/bin/perl*, and Perl will then dispatch the program to the correct interpreter for them.

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

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

#! and quoting on non-Unix systems

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

OS/2

Put

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

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

MS-DOS

Create a batch file to run your program, and codify it in `ALTERNATE_SHEBANG` (see the *dosish.h* file in the source distribution for more information).

Win95/NT

The Win95/NT installation, when using the ActiveState installer for Perl, will modify the Registry to associate the *.pl* extension with the perl interpreter. If you install Perl by other means (including building from the sources), you may have to modify the Registry yourself. Note that this means you can no longer tell the difference between an executable Perl program and a Perl library file.

VMS

Put

```
$ perl -mysw 'f$env("procedure")' 'p1' 'p2' 'p3' 'p4' 'p5' 'p6' 'p7' 'p8' !
$ exit++ + ++$status != 0 and $exit = $status = undef;
```

at the top of your program, where **-mysw** are any command line switches you want to pass to Perl. You can now invoke the program directly, by saying `perl program`, or as a DCL procedure, by saying `@program` (or implicitly via *DCL$PATH* by just using the name of the program).

This incantation is a bit much to remember, but Perl will display it for you if you say `perl "-V:startperl"`.

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

On some systems, you may have to change single-quotes to double ones, which you must *not* do on Unix or Plan 9 systems. You might also have to change a single % to a %%.

For example:

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

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

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

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

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

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

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

Location of Perl

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

In this documentation, `#!/usr/bin/perl` on the first line of the program will stand in for whatever method works on your system. You are advised to use a specific path if you care about a specific version.

```
#!/usr/local/bin/perl5.14
```

or if you just want to be running at least version, place a statement like this at the top of your program:

```
use 5.014;
```

Command Switches

As with all standard commands, a single-character switch may be clustered with the following switch, if any.

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

A `--` signals the end of options and disables further option processing. Any arguments after the `--` are treated as filenames and arguments.

Switches include:

-0[*octal/hexadecimal*]

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

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

The special value 00 will cause Perl to slurp files in paragraph mode. Any value 0400 or above will cause Perl to slurp files whole, but by convention the value 0777 is the one normally used for this purpose.

You can also specify the separator character using hexadecimal notation: **-0x*HHH...***, where the *H* are valid hexadecimal digits. Unlike the octal form, this one may be used to specify any Unicode character, even those beyond 0xFF. So if you *really* want a record separator of 0777, specify it as **-0x1FF**. (This means that you cannot use the **-x** option with a directory name that consists of hexadecimal digits, or else Perl will think you have specified a hex number to **-0**.)

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

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

is equivalent to

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

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

-a implicitly sets **-n**.

-C [*number/list*]

The **-C** flag controls some of the Perl Unicode features.

As of 5.8.1, the **-C** can be followed either by a number or a list of option letters. The letters, their numeric values, and effects are as follows; listing the letters is equal to summing the numbers.

```
I     1   STDIN is assumed to be in UTF-8
O     2   STDOUT will be in UTF-8
E     4   STDERR will be in UTF-8
S     7   I + O + E
i     8   UTF-8 is the default PerlIO layer for input streams
o    16   UTF-8 is the default PerlIO layer for output streams
D    24   i + o
A    32   the @ARGV elements are expected to be strings encoded
          in UTF-8
L    64   normally the "IOEioA" are unconditional, the L makes
          them conditional on the locale environment variables
          (the LC_ALL, LC_CTYPE, and LANG, in the order of
          decreasing precedence) -- if the variables indicate
          UTF-8, then the selected "IOEioA" are in effect
a   256   Set ${^UTF8CACHE} to -1, to run the UTF-8 caching
          code in debugging mode.
```

For example, **-COE** and **-C6** will both turn on UTF-8-ness on both STDOUT and STDERR. Repeating letters is just redundant, not cumulative nor toggling.

The `io` options mean that any subsequent *open()* (or similar I/O operations) in the current file scope will have the `:utf8` PerlIO layer implicitly applied to them, in other words, UTF-8 is expected from any input stream, and UTF-8 is produced to any output stream. This is just the default, with explicit layers in *open()* and with *binmode()* one can manipulate streams as usual.

-C on its own (not followed by any number or option list), or the empty string `""` for the `PERL_UNICODE` environment variable, has the same effect as **-CSDL**. In other words, the standard I/O handles and the default `open()` layer are UTF-8-fied *but* only if the locale environment variables indicate a UTF-8 locale. This behaviour follows the *implicit* (and problematic) UTF-8 behaviour of Perl 5.8.0. (See "UTF-8 no longer default under UTF-8 locales" in perl581delta.)

You can use **-C0** (or `"0"` for `PERL_UNICODE`) to explicitly disable all the above Unicode features.

The read-only magic variable `${^UNICODE}` reflects the numeric value of this setting. This variable is set during Perl startup and is thereafter read-only. If you want runtime effects, use the three-arg *open()* (see "open" in perlfunc), the two-arg *binmode()* (see "binmode" in perlfunc), and the `open` pragma (see open).

(In Perls earlier than 5.8.1 the **-C** switch was a Win32-only switch that enabled the use of Unicode-aware "wide system call" Win32 APIs. This feature was practically unused, however, and the command line switch was therefore "recycled".)

Note: Since perl 5.10.1, if the **-C** option is used on the `#!` line, it must be specified on the command line as well, since the standard streams are already set up at this point in the execution of the perl interpreter. You can also use *binmode()* to set the encoding of an I/O stream.

-c causes Perl to check the syntax of the program and then exit without executing it. Actually, it *will* execute any `BEGIN`, `UNITCHECK`, or `CHECK` blocks and any `use` statements: these are considered as occurring outside the execution of your program. `INIT` and `END` blocks, however, will be skipped.

-d

-dt runs the program under the Perl debugger. See perldebug. If **t** is specified, it indicates to the debugger that threads will be used in the code being debugged.

-d:*MOD[=bar,baz]*

-dt:*MOD[=bar,baz]*

runs the program under the control of a debugging, profiling, or tracing module installed as `Devel::`*MOD*. E.g., **-d:DProf** executes the program using the `Devel::DProf` profiler. As with the **-M** flag, options may be passed to the `Devel::`*MOD* package where they will be received and interpreted by the `Devel::`*MOD*`::import` routine. Again, like **-M**, use **--d:-*MOD*** to call `Devel::`*MOD*`::unimport` instead of import. The comma-separated list of options must follow a = character. If **t** is specified, it indicates to the debugger that threads will be used in the code being debugged. See perldebug.

–D*letters*
–D*number*

sets debugging flags. This switch is enabled only if your perl binary has been built with debugging enabled: normal production perls won't have been.

For example, to watch how perl executes your program, use **–Dtls**. Another nice value is **–Dx**, which lists your compiled syntax tree, and **–Dr** displays compiled regular expressions; the format of the output is explained in perldebguts.

As an alternative, specify a number instead of list of letters (e.g., **–D14** is equivalent to **–Dtls**):

```
        1  p  Tokenizing and parsing (with v, displays parse
              stack)
        2  s  Stack snapshots (with v, displays all stacks)
        4  l  Context (loop) stack processing
        8  t  Trace execution
       16  o  Method and overloading resolution
       32  c  String/numeric conversions
       64  P  Print profiling info, source file input state
      128  m  Memory and SV allocation
      256  f  Format processing
      512  r  Regular expression parsing and execution
     1024  x  Syntax tree dump
     2048  u  Tainting checks
     4096  U  Unofficial, User hacking (reserved for private,
              unreleased use)
     8192  H  Hash dump -- usurps values()
    16384  X  Scratchpad allocation
    32768  D  Cleaning up
    65536  S  Op slab allocation
   131072  T  Tokenizing
   262144  R  Include reference counts of dumped variables
              (eg when using -Ds)
   524288  J  show s,t,P-debug (don't Jump over) on opcodes within
              package DB
  1048576  v  Verbose: use in conjunction with other flags
  2097152  C  Copy On Write
  4194304  A  Consistency checks on internal structures
  8388608  q  quiet - currently only suppresses the "EXECUTING"
              message
 16777216  M  trace smart match resolution
 33554432  B  dump suBroutine definitions, including special
              Blocks like BEGIN
 67108864  L  trace Locale-related info; what gets output is very
              subject to change
134217728  i  trace PerlIO layer processing.  Set PERLIO_DEBUG to
              the filename to trace to.
```

All these flags require **–DDEBUGGING** when you compile the Perl executable (but see `:opd` in Devel::Peek or "'debug' mode" in re which may change this). See the *INSTALL* file in the Perl source distribution for how to do this.

If you're just trying to get a print out of each line of Perl code as it executes, the way that `sh -x` provides for shell scripts, you can't use Perl's **–D** switch. Instead do this

```
# If you have "env" utility
env PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

# Bourne shell syntax
$ PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

# csh syntax
```

```
% (setenv PERLDB_OPTS "NonStop=1 AutoTrace=1 frame=2"; perl -dS program)
```

See perldebug for details and variations.

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

−E *commandline*
: behaves just like **−e**, except that it implicitly enables all optional features (in the main compilation unit). See feature.

−f
: Disable executing *$Config{sitelib}/sitecustomize.pl* at startup.

  Perl can be built so that it by default will try to execute *$Config{sitelib}/sitecustomize.pl* at startup (in a BEGIN block). This is a hook that allows the sysadmin to customize how Perl behaves. It can for instance be used to add entries to the `@INC` array to make Perl find modules in non-standard locations.

  Perl actually inserts the following code:

```
BEGIN {
    do { local $!; -f "$Config{sitelib}/sitecustomize.pl"; }
        && do "$Config{sitelib}/sitecustomize.pl";
}
```

  Since it is an actual `do` (not a `require`), *sitecustomize.pl* doesn't need to return a true value. The code is run in package `main`, in its own lexical scope. However, if the script dies, `$@` will not be set.

  The value of `$Config{sitelib}` is also determined in C code and not read from `Config.pm`, which is not loaded.

  The code is executed *very* early. For example, any changes made to `@INC` will show up in the output of 'perl −V'. Of course, `END` blocks will be likewise executed very late.

  To determine at runtime if this capability has been compiled in your perl, you can check the value of `$Config{usesitecustomize}`.

−F*pattern*
: specifies the pattern to split on for **−a**. The pattern may be surrounded by `//`, `""`, or `''`, otherwise it will be put in single quotes. You can't use literal whitespace or NUL characters in the pattern.

  −F implicitly sets both **−a** and **−n**.

−h
: prints a summary of the options.

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

  If no extension is supplied, and your system supports it, the original *file* is kept open without a name while the output is redirected to a new file with the original *filename*. When perl exits, cleanly or not, the original *file* is unlinked.

  If the extension doesn't contain a `*`, then it is appended to the end of the current filename as a suffix. If the extension does contain one or more `*` characters, then each `*` is replaced with the current filename. In Perl terms, you could think of this as:

```
($backup = $extension) =~ s/\*/$file_name/g;
```

  This allows you to add a prefix to the backup file, instead of (or in addition to) a suffix:

```
 $ perl -pi'orig_*' -e 's/bar/baz/' fileA  # backup to
                                           # 'orig_fileA'
```

  Or even to place backup copies of the original files into another directory (provided the directory already exists):

```
 $ perl -pi'old/*.orig' -e 's/bar/baz/' fileA  # backup to
                                               # 'old/fileA.orig'
```

These sets of one-liners are equivalent:

```
 $ perl -pi -e 's/bar/baz/' fileA          # overwrite current file
 $ perl -pi'*' -e 's/bar/baz/' fileA       # overwrite current file

 $ perl -pi'.orig' -e 's/bar/baz/' fileA   # backup to 'fileA.orig'
 $ perl -pi'*.orig' -e 's/bar/baz/' fileA  # backup to 'fileA.orig'
```

From the shell, saying

```
    $ perl -p -i.orig -e "s/foo/bar/; ... "
```

is the same as using the program:

```
    #!/usr/bin/perl -pi.orig
    s/foo/bar/;
```

which is equivalent to

```
    #!/usr/bin/perl
    $extension = '.orig';
    LINE: while (<>) {
        if ($ARGV ne $oldargv) {
            if ($extension !~ /\*/) {
                $backup = $ARGV . $extension;
            }
            else {
                ($backup = $extension) =~ s/\*/$ARGV/g;
            }
            rename($ARGV, $backup);
            open(ARGVOUT, ">$ARGV");
            select(ARGVOUT);
            $oldargv = $ARGV;
        }
        s/foo/bar/;
    }
    continue {
        print;  # this prints to original filename
    }
    select(STDOUT);
```

except that the **–i** form doesn't need to compare `$ARGV` to `$oldargv` to know when the filename has changed. It does, however, use ARGVOUT for the selected filehandle. Note that STDOUT is restored as the default output filehandle after the loop.

As shown above, Perl creates the backup file whether or not any output is actually changed. So this is just a fancy way to copy files:

```
    $ perl -p -i'/some/file/path/*' -e 1 file1 file2 file3...
or
    $ perl -p -i'.orig' -e 1 file1 file2 file3...
```

You can use `eof` without parentheses to locate the end of each input file, in case you want to append to each file, or reset line numbering (see example in "eof" in perlfunc).

If, for a given file, Perl is unable to create the backup file as specified in the extension then it will skip that file and continue on with the next one (if it exists).

For a discussion of issues surrounding file permissions and **–i**, see "Why does Perl let me delete read-only files? Why does –i clobber protected files? Isn't this a bug in Perl?" in perlfaq5.

You cannot use **–i** to create directories or to strip extensions from files.

Perl does not expand ~ in filenames, which is good, since some folks use it for their backup files:

```
$ perl -pi~ -e 's/foo/bar/' file1 file2 file3...
```

Note that because **-i** renames or deletes the original file before creating a new file of the same name, Unix-style soft and hard links will not be preserved.

Finally, the **-i** switch does not impede execution when no files are given on the command line. In this case, no backup is made (the original file cannot, of course, be determined) and processing proceeds from STDIN to STDOUT as might be expected.

-I*directory*
Directories specified by **-I** are prepended to the search path for modules (`@INC`).

-l[*octnum*]
enables automatic line-ending processing. It has two separate effects. First, it automatically chomps `$/` (the input record separator) when used with **-n** or **-p**. Second, it assigns `$\` (the output record separator) to have the value of *octnum* so that any print statements will have that separator added back on. If *octnum* is omitted, sets `$\` to the current value of `$/`. For instance, to trim lines to 80 columns:

```
perl -lpe 'substr($_, 80) = ""'
```

Note that the assignment `$\ = $/` is done when the switch is processed, so the input record separator can be different than the output record separator if the **-l** switch is followed by a **-0** switch:

```
gnufind / -print0 | perl -ln0e 'print "found $_" if -p'
```

This sets `$\` to newline and then sets `$/` to the null character.

-m[**-**]*module*
-M[**-**]*module*
-M[**-**]*'module ...'*
-[mM][**-**]*module=arg[,arg]...*
-m*module* executes `use` *module* `();` before executing your program.

-M*module* executes `use` *module* `;` before executing your program. You can use quotes to add extra code after the module name, e.g., `'-M`*`MODULE`*` qw(foo bar)'`.

If the first character after the **-M** or **-m** is a dash (**-**) then the 'use' is replaced with 'no'.

A little builtin syntactic sugar means you can also say **-m*MODULE*=foo,bar** or **-M*MODULE*=foo,bar** as a shortcut for **'-M*MODULE* qw(foo bar)'**. This avoids the need to use quotes when importing symbols. The actual code generated by **-M*MODULE*=foo,bar** is `use module split(/,/,q{foo,bar})`. Note that the = form removes the distinction between **-m** and **-M**; that is, **-m*MODULE*=foo,bar** is the same as **-M*MODULE*=foo,bar**.

A consequence of this is that **-M*MODULE*=number** never does a version check, unless *MODULE*`::import()` itself is set up to do a version check, which could happen for example if *MODULE* inherits from Exporter.

-n causes Perl to assume the following loop around your program, which makes it iterate over filename arguments somewhat like *sed -n* or *awk*:

```
LINE:
  while (<>) {
      ...             # your program goes here
  }
```

Note that the lines are not printed by default. See "-p" to have lines printed. If a file named by an argument cannot be opened for some reason, Perl warns you about it and moves on to the next file.

Also note that <> passes command line arguments to "open" in perlfunc, which doesn't necessarily interpret them as file names. See perlop for possible security implications.

Here is an efficient way to delete all files that haven't been modified for at least a week:

```
find . -mtime +7 -print | perl -nle unlink
```

This is faster than using the **-exec** switch of *find* because you don't have to start a process on every filename found (but it's not faster than using the **-delete** switch available in newer versions of *find*. It

does suffer from the bug of mishandling newlines in pathnames, which you can fix if you follow the example under **−0**.

`BEGIN` and `END` blocks may be used to capture control before or after the implicit program loop, just as in *awk*.

−p causes Perl to assume the following loop around your program, which makes it iterate over filename arguments somewhat like *sed*:

```
LINE:
  while (<>) {
      ...             # your program goes here
  } continue {
      print or die "-p destination: $!\n";
  }
```

If a file named by an argument cannot be opened for some reason, Perl warns you about it, and moves on to the next file. Note that the lines are printed automatically. An error occurring during printing is treated as fatal. To suppress printing use the **−n** switch. A **−p** overrides a **−n** switch.

`BEGIN` and `END` blocks may be used to capture control before or after the implicit loop, just as in *awk*.

−s enables rudimentary switch parsing for switches on the command line after the program name but before any filename arguments (or before an argument of **−−**). Any switch found there is removed from `@ARGV` and sets the corresponding variable in the Perl program. The following program prints "1" if the program is invoked with a **−xyz** switch, and "abc" if it is invoked with **−xyz=abc**.

```
#!/usr/bin/perl -s
if ($xyz) { print "$xyz\n" }
```

Do note that a switch like **−−help** creates the variable `${-help}`, which is not compliant with `use strict "refs"`. Also, when using this option on a script with warnings enabled you may get a lot of spurious "used only once" warnings.

−S makes Perl use the PATH environment variable to search for the program unless the name of the program contains path separators.

On some platforms, this also makes Perl append suffixes to the filename while searching for it. For example, on Win32 platforms, the ".bat" and ".cmd" suffixes are appended if a lookup for the original name fails, and if the name does not already end in one of those suffixes. If your Perl was compiled with `DEBUGGING` turned on, using the **−Dp** switch to Perl shows how the search progresses.

Typically this is used to emulate `#!` startup on platforms that don't support `#!`. It's also convenient when debugging a script that uses `#!`, and is thus normally found by the shell's `$PATH` search mechanism.

This example works on many platforms that have a shell compatible with Bourne shell:

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

The system ignores the first line and feeds the program to */bin/sh*, which proceeds to try to execute the Perl program as a shell script. The shell executes the second line as a normal shell command, and thus starts up the Perl interpreter. On some systems `$0` doesn't always contain the full pathname, so the **−S** tells Perl to search for the program if necessary. After Perl locates the program, it parses the lines and ignores them because the variable `$running_under_some_shell` is never true. If the program will be interpreted by csh, you will need to replace `${1+"$@"}` with `$*`, even though that doesn't understand embedded spaces (and such) in the argument list. To start up *sh* rather than *csh*, some systems may have to replace the `#!` line with a line containing just a colon, which will be politely ignored by Perl. Other systems can't control that, and need a totally devious construct that will work under any of *csh*, *sh*, or Perl, such as the following:

```
    eval '(exit $?0)' && eval 'exec perl -wS $0 ${1+"$@"}'
    & eval 'exec /usr/bin/perl -wS $0 $argv:q'
            if $running_under_some_shell;
```

If the filename supplied contains directory separators (and so is an absolute or relative pathname), and if that file is not found, platforms that append file extensions will do so and try to look for the file with those extensions added, one by one.

On DOS-like platforms, if the program does not contain directory separators, it will first be searched for in the current directory before being searched for on the PATH. On Unix platforms, the program will be searched for strictly on the PATH.

-t Like **-T**, but taint checks will issue warnings rather than fatal errors. These warnings can now be controlled normally with `no warnings qw(taint)`.

Note: This is not a substitute for `-T`! This is meant to be used *only* as a temporary development aid while securing legacy code: for real production code and for new secure code written from scratch, always use the real **-T**.

-T turns on "taint" so you can test them. Ordinarily these checks are done only when running setuid or setgid. It's a good idea to turn them on explicitly for programs that run on behalf of someone else whom you might not necessarily trust, such as CGI programs or any internet servers you might write in Perl. See perlsec for details. For security reasons, this option must be seen by Perl quite early; usually this means it must appear early on the command line or in the `#!` line for systems which support that construct.

-u This switch causes Perl to dump core after compiling your program. You can then in theory take this core dump and turn it into an executable file by using the *undump* program (not supplied). This speeds startup at the expense of some disk space (which you can minimize by stripping the executable). (Still, a "hello world" executable comes out to about 200K on my machine.) If you want to execute a portion of your program before dumping, use the *dump()* operator instead. Note: availability of *undump* is platform specific and may not be available for a specific port of Perl.

-U allows Perl to do unsafe operations. Currently the only "unsafe" operations are attempting to unlink directories while running as superuser and running setuid programs with fatal taint checks turned into warnings. Note that warnings must be enabled along with this option to actually *generate* the taint-check warnings.

-v prints the version and patchlevel of your perl executable.

-V prints summary of the major perl configuration values and the current values of `@INC`.

-V:*configvar*

Prints to STDOUT the value of the named configuration variable(s), with multiples when your *`configvar`* argument looks like a regex (has non-letters). For example:

```
$ perl -V:libc
    libc='/lib/libc-2.2.4.so';
$ perl -V:lib.
    libs='-lnsl -lgdbm -ldb -ldl -lm -lcrypt -lutil -lc';
    libc='/lib/libc-2.2.4.so';
$ perl -V:lib.*
    libpth='/usr/local/lib /lib /usr/lib';
    libs='-lnsl -lgdbm -ldb -ldl -lm -lcrypt -lutil -lc';
    lib_ext='.a';
    libc='/lib/libc-2.2.4.so';
    libperl='libperl.a';
    ....
```

Additionally, extra colons can be used to control formatting. A trailing colon suppresses the linefeed and terminator ";", allowing you to embed queries into shell commands. (mnemonic: PATH separator ":".)

```
$ echo "compression-vars: " `perl -V:z.*: ` " are here !"
compression-vars:  zcat='' zip='zip'  are here !
```

A leading colon removes the "name=" part of the response, this allows you to map to the name you need. (mnemonic: empty label)

```
$ echo "goodvfork="`./perl -Ilib -V::usevfork`
goodvfork=false;
```

Leading and trailing colons can be used together if you need positional parameter values without the names. Note that in the case below, the `PERL_API` params are returned in alphabetical order.

```
$ echo building_on `perl -V::osname: -V::PERL_API_.*:` now
building_on 'linux' '5' '1' '9' now
```

-w prints warnings about dubious constructs, such as variable names mentioned only once and scalar variables used before being set; redefined subroutines; references to undefined filehandles; filehandles opened read-only that you are attempting to write on; values used as a number that don't *look* like numbers; using an array as though it were a scalar; if your subroutines recurse more than 100 deep; and innumerable other things.

This switch really just enables the global `$^W` variable; normally, the lexically scoped `use warnings` pragma is preferred. You can disable or promote into fatal errors specific warnings using `__WARN__` hooks, as described in perlvar and "warn" in perlfunc. See also perldiag and perltrap. A fine-grained warning facility is also available if you want to manipulate entire classes of warnings; see warnings.

-W Enables all warnings regardless of `no warnings` or `$^W`. See warnings.

-X Disables all warnings regardless of `use warnings` or `$^W`. See warnings.

-x
-x*directory*

tells Perl that the program is embedded in a larger chunk of unrelated text, such as in a mail message. Leading garbage will be discarded until the first line that starts with `#!` and contains the string "perl". Any meaningful switches on that line will be applied.

All references to line numbers by the program (warnings, errors, ...) will treat the `#!` line as the first line. Thus a warning on the 2nd line of the program, which is on the 100th line in the file will be reported as line 2, not as line 100. This can be overridden by using the `#line` directive. (See "Plain Old Comments (Not!)" in perlsyn)

If a directory name is specified, Perl will switch to that directory before running the program. The **-x** switch controls only the disposal of leading garbage. The program must be terminated with `__END__` if there is trailing garbage to be ignored; the program can process any or all of the trailing garbage via the `DATA` filehandle if desired.

The directory, if specified, must appear immediately following the **-x** with no intervening whitespace.

ENVIRONMENT

HOME Used if `chdir` has no argument.

LOGDIR Used if `chdir` has no argument and HOME is not set.

PATH Used in executing subprocesses, and in finding the program if **-S** is used.

PERL5LIB A list of directories in which to look for Perl library files before looking in the standard library and the current directory. Any architecture-specific and version-specific directories, such as *version/archname/*, *version/*, or *archname/* under the specified locations are automatically included if they exist, with this lookup done at interpreter startup time. In addition, any directories matching the entries in `$Config{inc_version_list}` are added. (These typically would be for older compatible perl versions installed in the same directory tree.)

If PERL5LIB is not defined, PERLLIB is used. Directories are separated (like in PATH) by a colon on Unixish platforms and by a semicolon on Windows (the proper path separator being given by the command `perl -V:path_sep`).

When running taint checks, either because the program was running setuid or setgid, or the **-T** or **-t** switch was specified, neither PERL5LIB nor PERLLIB is consulted. The program

should instead say:

```
use lib "/my/directory";
```

PERL5OPT Command-line options (switches). Switches in this variable are treated as if they were on every Perl command line. Only the **−[CDIMUdmtwW]** switches are allowed. When running taint checks (either because the program was running setuid or setgid, or because the **−T** or **−t** switch was used), this variable is ignored. If PERL5OPT begins with **−T**, tainting will be enabled and subsequent options ignored. If PERL5OPT begins with **−t**, tainting will be enabled, a writable dot removed from `@INC`, and subsequent options honored.

PERLIO A space (or colon) separated list of PerlIO layers. If perl is built to use PerlIO system for IO (the default) these layers affect Perl's IO.

It is conventional to start layer names with a colon (for example, `:perlio`) to emphasize their similarity to variable "attributes". But the code that parses layer specification strings, which is also used to decode the PERLIO environment variable, treats the colon as a separator.

An unset or empty PERLIO is equivalent to the default set of layers for your platform; for example, `:unix:perlio` on Unix-like systems and `:unix:crlf` on Windows and other DOS-like systems.

The list becomes the default for *all* Perl's IO. Consequently only built-in layers can appear in this list, as external layers (such as `:encoding()`) need IO in order to load them! See "open pragma" for how to add external encodings as defaults.

Layers it makes sense to include in the PERLIO environment variable are briefly summarized below. For more details see PerlIO.

:bytes A pseudolayer that turns the `:utf8` flag *off* for the layer below; unlikely to be useful on its own in the global PERLIO environment variable. You perhaps were thinking of `:crlf:bytes` or `:perlio:bytes`.

:crlf A layer which does CRLF to `"\n"` translation distinguishing "text" and "binary" files in the manner of MS-DOS and similar operating systems. (It currently does *not* mimic MS-DOS as far as treating of Control-Z as being an end-of-file marker.)

:mmap A layer that implements "reading" of files by using *mmap*(2) to make an entire file appear in the process's address space, and then using that as PerlIO's "buffer".

:perlio This is a re-implementation of stdio-like buffering written as a PerlIO layer. As such it will call whatever layer is below it for its operations, typically `:unix`.

:pop An experimental pseudolayer that removes the topmost layer. Use with the same care as is reserved for nitroglycerine.

:raw A pseudolayer that manipulates other layers. Applying the `:raw` layer is equivalent to calling `binmode($fh)`. It makes the stream pass each byte as-is without translation. In particular, both CRLF translation and intuiting `:utf8` from the locale are disabled.

Unlike in earlier versions of Perl, `:raw` is *not* just the inverse of `:crlf`: other layers which would affect the binary nature of the stream are also removed or disabled.

:stdio This layer provides a PerlIO interface by wrapping system's ANSI C "stdio" library calls. The layer provides both buffering and IO. Note that the `:stdio` layer does *not* do CRLF translation even if that is the platform's normal behaviour. You will need a `:crlf` layer above it to do that.

:unix Low-level layer that calls `read`, `write`, `lseek`, etc.

:utf8 A pseudolayer that enables a flag in the layer below to tell Perl that output should be in utf8 and that input should be regarded as already in valid utf8 form. **WARNING: It does not check for validity and as such should be handled with extreme caution for input, because security violations can occur with non-**

shortest UTF-8 encodings, etc. Generally `:encoding(UTF-8)` is the best option when reading UTF-8 encoded data.

:win32 On Win32 platforms this *experimental* layer uses native "handle" IO rather than a Unix-like numeric file descriptor layer. Known to be buggy in this release (5.14).

The default set of layers should give acceptable results on all platforms

For Unix platforms that will be the equivalent of "unix perlio" or "stdio". Configure is set up to prefer the "stdio" implementation if the system's library provides for fast access to the buffer; otherwise, it uses the "unix perlio" implementation.

On Win32 the default in this release (5.14) is "unix crlf". Win32's "stdio" has a number of bugs/mis-features for Perl IO which are somewhat depending on the version and vendor of the C compiler. Using our own `crlf` layer as the buffer avoids those issues and makes things more uniform. The `crlf` layer provides CRLF conversion as well as buffering.

This release (5.14) uses `unix` as the bottom layer on Win32, and so still uses the C compiler's numeric file descriptor routines. There is an experimental native `win32` layer, which is expected to be enhanced and should eventually become the default under Win32.

The PERLIO environment variable is completely ignored when Perl is run in taint mode.

PERLIO_DEBUG

If set to the name of a file or device when Perl is run with the **-Di** command-line switch, the logging of certain operations of the PerlIO subsystem will be redirected to the specified file rather than going to stderr, which is the default. The file is opened in append mode. Typical uses are in Unix:

```
% env PERLIO_DEBUG=/tmp/perlio.log perl -Di script ...
```

and under Win32, the approximately equivalent:

```
> set PERLIO_DEBUG=CON
perl -Di script ...
```

This functionality is disabled for setuid scripts, for scripts run with **-T**, and for scripts run on a Perl built without `-DDEBUGGING` support.

PERLLIB A list of directories in which to look for Perl library files before looking in the standard library and the current directory. If PERL5LIB is defined, PERLLIB is not used.

The PERLLIB environment variable is completely ignored when Perl is run in taint mode.

PERL5DB The command used to load the debugger code. The default is:

```
BEGIN { require "perl5db.pl" }
```

The PERL5DB environment variable is only used when Perl is started with a bare **-d** switch.

PERL5DB_THREADED

If set to a true value, indicates to the debugger that the code being debugged uses threads.

PERL5SHELL (specific to the Win32 port)

On Win32 ports only, may be set to an alternative shell that Perl must use internally for executing "backtick" commands or *system()*. Default is `cmd.exe /x/d/c` on WindowsNT and `command.com /c` on Windows95. The value is considered space-separated. Precede any character that needs to be protected, like a space or backslash, with another backslash.

Note that Perl doesn't use COMSPEC for this purpose because COMSPEC has a high degree of variability among users, leading to portability concerns. Besides, Perl can use a shell that may not be fit for interactive use, and setting COMSPEC to such a shell may interfere with the proper functioning of other programs (which usually look in COMSPEC to find a shell fit for interactive use).

Before Perl 5.10.0 and 5.8.8, PERL5SHELL was not taint checked when running external commands. It is recommended that you explicitly set (or delete) `$ENV{PERL5SHELL}` when running in taint mode under Windows.

PERL_ALLOW_NON_IFS_LSP (specific to the Win32 port)

Set to 1 to allow the use of non-IFS compatible LSPs (Layered Service Providers). Perl normally searches for an IFS-compatible LSP because this is required for its emulation of Windows sockets as real filehandles. However, this may cause problems if you have a firewall such as *McAfee Guardian*, which requires that all applications use its LSP but which is not IFS-compatible, because clearly Perl will normally avoid using such an LSP.

Setting this environment variable to 1 means that Perl will simply use the first suitable LSP enumerated in the catalog, which keeps *McAfee Guardian* happy — and in that particular case Perl still works too because *McAfee Guardian*'s LSP actually plays other games which allow applications requiring IFS compatibility to work.

PERL_DEBUG_MSTATS

Relevant only if Perl is compiled with the `malloc` included with the Perl distribution; that is, if `perl -V:d_mymalloc` is "define".

If set, this dumps out memory statistics after execution. If set to an integer greater than one, also dumps out memory statistics after compilation.

PERL_DESTRUCT_LEVEL

Relevant only if your Perl executable was built with **-DDEBUGGING**, this controls the behaviour of global destruction of objects and other references. See "PERL_DESTRUCT_LEVEL" in perlhacktips for more information.

PERL_DL_NONLAZY

Set to `"1"` to have Perl resolve *all* undefined symbols when it loads a dynamic library. The default behaviour is to resolve symbols when they are used. Setting this variable is useful during testing of extensions, as it ensures that you get an error on misspelled function names even if the test suite doesn't call them.

PERL_ENCODING

If using the `use encoding` pragma without an explicit encoding name, the PERL_ENCODING environment variable is consulted for an encoding name.

PERL_HASH_SEED

(Since Perl 5.8.1, new semantics in Perl 5.18.0) Used to override the randomization of Perl's internal hash function. The value is expressed in hexadecimal, and may include a leading 0x. Truncated patterns are treated as though they are suffixed with sufficient 0's as required.

If the option is provided, and `PERL_PERTURB_KEYS` is NOT set, then a value of '0' implies `PERL_PERTURB_KEYS=0` and any other value implies `PERL_PERTURB_KEYS=2`.

PLEASE NOTE: The hash seed is sensitive information. Hashes are randomized to protect against local and remote attacks against Perl code. By manually setting a seed, this protection may be partially or completely lost.

See "Algorithmic Complexity Attacks" in perlsec, "PERL_PERTURB_KEYS", and "PERL_HASH_SEED_DEBUG" for more information.

PERL_PERTURB_KEYS

(Since Perl 5.18.0) Set to `"0"` or `"NO"` then traversing keys will be repeatable from run to run for the same PERL_HASH_SEED. Insertion into a hash will not change the order, except to provide for more space in the hash. When combined with setting PERL_HASH_SEED this mode is as close to pre 5.18 behavior as you can get.

When set to `"1"` or `"RANDOM"` then traversing keys will be randomized. Every time a hash is inserted into the key order will change in a random fashion. The order may not be repeatable in a following program run even if the PERL_HASH_SEED has been specified. This is the default mode for perl.

When set to `"2"` or `"DETERMINISTIC"` then inserting keys into a hash will cause the key order to change, but in a way that is repeatable from program run to program run.

NOTE: Use of this option is considered insecure, and is intended only for debugging non-

deterministic behavior in Perl's hash function. Do not use it in production.

See "Algorithmic Complexity Attacks" in perlsec and "PERL_HASH_SEED" and "PERL_HASH_SEED_DEBUG" for more information. You can get and set the key traversal mask for a specific hash by using the `hash_traversal_mask()` function from Hash::Util.

PERL_HASH_SEED_DEBUG

(Since Perl 5.8.1.) Set to `"1"` to display (to STDERR) information about the hash function, seed, and what type of key traversal randomization is in effect at the beginning of execution. This, combined with "PERL_HASH_SEED" and "PERL_PERTURB_KEYS" is intended to aid in debugging nondeterministic behaviour caused by hash randomization.

Note that any information about the hash function, especially the hash seed is **sensitive information**: by knowing it, one can craft a denial-of-service attack against Perl code, even remotely; see "Algorithmic Complexity Attacks" in perlsec for more information. **Do not disclose the hash seed** to people who don't need to know it. See also `hash_seed()` and `key_traversal_mask()` in Hash::Util.

An example output might be:

```
HASH_FUNCTION = ONE_AT_A_TIME_HARD HASH_SEED = 0x652e9b9349a7a032 PERT
```

PERL_MEM_LOG

If your Perl was configured with **-Accflags=-DPERL_MEM_LOG**, setting the environment variable `PERL_MEM_LOG` enables logging debug messages. The value has the form `<`*number*`>[m][s][t]`, where *number* is the file descriptor number you want to write to (2 is default), and the combination of letters specifies that you want information about (m)emory and/or (s)v, optionally with (t)imestamps. For example, `PERL_MEM_LOG=1mst` logs all information to stdout. You can write to other opened file descriptors in a variety of ways:

```
$ 3>foo3 PERL_MEM_LOG=3m perl ...
```

PERL_ROOT (specific to the VMS port)

A translation-concealed rooted logical name that contains Perl and the logical device for the `@INC` path on VMS only. Other logical names that affect Perl on VMS include PERLSHR, PERL_ENV_TABLES, and SYS$TIMEZONE_DIFFERENTIAL, but are optional and discussed further in perlvms and in *README.vms* in the Perl source distribution.

PERL_SIGNALS

Available in Perls 5.8.1 and later. If set to `"unsafe"`, the pre-Perl-5.8.0 signal behaviour (which is immediate but unsafe) is restored. If set to `safe`, then safe (but deferred) signals are used. See "Deferred Signals (Safe Signals)" in perlipc.

PERL_UNICODE

Equivalent to the **-C** command-line switch. Note that this is not a boolean variable. Setting this to `"1"` is not the right way to "enable Unicode" (whatever that would mean). You can use `"0"` to "disable Unicode", though (or alternatively unset PERL_UNICODE in your shell before starting Perl). See the description of the **-C** switch for more information.

PERL_USE_UNSAFE_INC

If perl has been configured to not have the current directory in `@INC` by default, this variable can be set to `"1"` to reinstate it. It's primarily intended for use while building and testing modules that have not been updated to deal with "." not being in `@INC` and should not be set in the environment for day-to-day use.

SYS$LOGIN (specific to the VMS port)

Used if chdir has no argument and HOME and LOGDIR are not set.

Perl also has environment variables that control how Perl handles data specific to particular natural languages; see perllocale.

Perl and its various modules and components, including its test frameworks, may sometimes make use of certain other environment variables. Some of these are specific to a particular platform. Please consult the appropriate module documentation and any documentation for your platform (like perlsolaris, perllinux, perlmacosx, perlwin32, etc) for variables peculiar to those specific situations.

Perl makes all environment variables available to the program being executed, and passes these along to any child processes it starts. However, programs running setuid would do well to execute the following lines before doing anything else, just to keep people honest:

```
$ENV{PATH}  = "/bin:/usr/bin";    # or whatever you need
$ENV{SHELL} = "/bin/sh" if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

NAME

perlreftut – Mark's very short tutorial about references

DESCRIPTION

One of the most important new features in Perl 5 was the capability to manage complicated data structures like multidimensional arrays and nested hashes. To enable these, Perl 5 introduced a feature called *references*, and using references is the key to managing complicated, structured data in Perl. Unfortunately, there's a lot of funny syntax to learn, and the main manual page can be hard to follow. The manual is quite complete, and sometimes people find that a problem, because it can be hard to tell what is important and what isn't.

Fortunately, you only need to know 10% of what's in the main page to get 90% of the benefit. This page will show you that 10%.

Who Needs Complicated Data Structures?

One problem that comes up all the time is needing a hash whose values are lists. Perl has hashes, of course, but the values have to be scalars; they can't be lists.

Why would you want a hash of lists? Let's take a simple example: You have a file of city and country names, like this:

```
Chicago, USA
Frankfurt, Germany
Berlin, Germany
Washington, USA
Helsinki, Finland
New York, USA
```

and you want to produce an output like this, with each country mentioned once, and then an alphabetical list of the cities in that country:

```
Finland: Helsinki.
Germany: Berlin, Frankfurt.
USA:  Chicago, New York, Washington.
```

The natural way to do this is to have a hash whose keys are country names. Associated with each country name key is a list of the cities in that country. Each time you read a line of input, split it into a country and a city, look up the list of cities already known to be in that country, and append the new city to the list. When you're done reading the input, iterate over the hash as usual, sorting each list of cities before you print it out.

If hash values couldn't be lists, you lose. You'd probably have to combine all the cities into a single string somehow, and then when time came to write the output, you'd have to break the string into a list, sort the list, and turn it back into a string. This is messy and error-prone. And it's frustrating, because Perl already has perfectly good lists that would solve the problem if only you could use them.

The Solution

By the time Perl 5 rolled around, we were already stuck with this design: Hash values must be scalars. The solution to this is references.

A reference is a scalar value that *refers to* an entire array or an entire hash (or to just about anything else). Names are one kind of reference that you're already familiar with. Think of the President of the United States: a messy, inconvenient bag of blood and bones. But to talk about him, or to represent him in a computer program, all you need is the easy, convenient scalar string ''Barack Obama''.

References in Perl are like names for arrays and hashes. They're Perl's private, internal names, so you can be sure they're unambiguous. Unlike ''Barack Obama'', a reference only refers to one thing, and you always know what it refers to. If you have a reference to an array, you can recover the entire array from it. If you have a reference to a hash, you can recover the entire hash. But the reference is still an easy, compact scalar value.

You can't have a hash whose values are arrays; hash values can only be scalars. We're stuck with that. But a single reference can refer to an entire array, and references are scalars, so you can have a hash of references to arrays, and it'll act a lot like a hash of arrays, and it'll be just as useful as a hash of arrays.

We'll come back to this city-country problem later, after we've seen some syntax for managing references.

Syntax

There are just two ways to make a reference, and just two ways to use it once you have it.

Making References

Make Rule 1

If you put a \ in front of a variable, you get a reference to that variable.

```
$aref = \@array;          # $aref now holds a reference to @array
$href = \%hash;           # $href now holds a reference to %hash
$sref = \$scalar;         # $sref now holds a reference to $scalar
```

Once the reference is stored in a variable like `$aref` or `$href`, you can copy it or store it just the same as any other scalar value:

```
$xy = $aref;              # $xy now holds a reference to @array
$p[3] = $href;            # $p[3] now holds a reference to %hash
$z = $p[3];               # $z now holds a reference to %hash
```

These examples show how to make references to variables with names. Sometimes you want to make an array or a hash that doesn't have a name. This is analogous to the way you like to be able to use the string `"\n"` or the number 80 without having to store it in a named variable first.

Make Rule 2

`[ ITEMS ]` makes a new, anonymous array, and returns a reference to that array. `{ ITEMS }` makes a new, anonymous hash, and returns a reference to that hash.

```
$aref = [ 1, "foo", undef, 13 ];
# $aref now holds a reference to an array

$href = { APR => 4, AUG => 8 };
# $href now holds a reference to a hash
```

The references you get from rule 2 are the same kind of references that you get from rule 1:

```
# This:
$aref = [ 1, 2, 3 ];

# Does the same as this:
@array = (1, 2, 3);
$aref = \@array;
```

The first line is an abbreviation for the following two lines, except that it doesn't create the superfluous array variable `@array`.

If you write just `[]`, you get a new, empty anonymous array. If you write just `{}`, you get a new, empty anonymous hash.

Using References

What can you do with a reference once you have it? It's a scalar value, and we've seen that you can store it as a scalar and get it back again just like any scalar. There are just two more ways to use it:

Use Rule 1

You can always use an array reference, in curly braces, in place of the name of an array. For example, `@{$aref}` instead of `@array`.

Here are some examples of that:

Arrays:

```
@a              @{$aref}                An array
reverse @a      reverse @{$aref}        Reverse the array
$a[3]           ${$aref}[3]             An element of the array
$a[3] = 17;     ${$aref}[3] = 17        Assigning an element
```

On each line are two expressions that do the same thing. The left-hand versions operate on the array `@a`. The right-hand versions operate on the array that is referred to by `$aref`. Once they find the array they're operating on, both versions do the same things to the arrays.

Using a hash reference is *exactly* the same:

```
%h               %{$href}               A hash
keys %h          keys %{$href}          Get the keys from the hash
$h{'red'}        ${$href}{'red'}        An element of the hash
$h{'red'} = 17   ${$href}{'red'} = 17   Assigning an element
```

Whatever you want to do with a reference, **Use Rule 1** tells you how to do it. You just write the Perl code that you would have written for doing the same thing to a regular array or hash, and then replace the array or hash name with `{$reference}`. "How do I loop over an array when all I have is a reference?" Well, to loop over an array, you would write

```
for my $element (@array) {
  ...
}
```

so replace the array name, `@array`, with the reference:

```
for my $element (@{$aref}) {
  ...
}
```

"How do I print out the contents of a hash when all I have is a reference?" First write the code for printing out a hash:

```
for my $key (keys %hash) {
  print "$key => $hash{$key}\n";
}
```

And then replace the hash name with the reference:

```
for my $key (keys %{$href}) {
  print "$key => ${$href}{$key}\n";
}
```

Use Rule 2

Use Rule 1 is all you really need, because it tells you how to do absolutely everything you ever need to do with references. But the most common thing to do with an array or a hash is to extract a single element, and the **Use Rule 1** notation is cumbersome. So there is an abbreviation.

`${$aref}[3]` is too hard to read, so you can write `$aref->[3]` instead.

`${$href}{red}` is too hard to read, so you can write `$href->{red}` instead.

If `$aref` holds a reference to an array, then `$aref->[3]` is the fourth element of the array. Don't confuse this with `$aref[3]`, which is the fourth element of a totally different array, one deceptively named `@aref`. `$aref` and `@aref` are unrelated the same way that `$item` and `@item` are.

Similarly, `$href->{'red'}` is part of the hash referred to by the scalar variable `$href`, perhaps even one with no name. `$href{'red'}` is part of the deceptively named `%href` hash. It's easy to forget to leave out the `->`, and if you do, you'll get bizarre results when your program gets array and hash elements out of totally unexpected hashes and arrays that weren't the ones you wanted to use.

An Example

Let's see a quick example of how all this is useful.

First, remember that `[1, 2, 3]` makes an anonymous array containing `(1, 2, 3)`, and gives you a reference to that array.

Now think about

```
@a = ( [1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]
     );
```

`@a` is an array with three elements, and each one is a reference to another array.

`$a[1]` is one of these references. It refers to an array, the array containing `(4, 5, 6)`, and because it is a reference to an array, **Use Rule 2** says that we can write `$a[1]->[2]` to get the third element from that

array. `$a[1]->[2]` is the 6. Similarly, `$a[0]->[1]` is the 2. What we have here is like a two-dimensional array; you can write `$a[ROW]->[COLUMN]` to get or set the element in any row and any column of the array.

The notation still looks a little cumbersome, so there's one more abbreviation:

Arrow Rule

In between two **subscripts**, the arrow is optional.

Instead of `$a[1]->[2]`, we can write `$a[1][2]`; it means the same thing. Instead of `$a[0]->[1] = 23`, we can write `$a[0][1] = 23`; it means the same thing.

Now it really looks like two-dimensional arrays!

You can see why the arrows are important. Without them, we would have had to write `${$a[1]}[2]` instead of `$a[1][2]`. For three-dimensional arrays, they let us write `$x[2][3][5]` instead of the unreadable `${${$x[2]}[3]}[5]`.

Solution

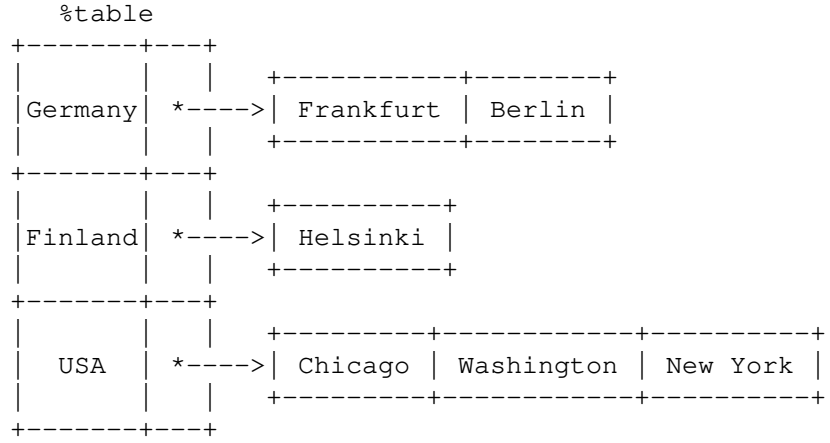
Here's the answer to the problem I posed earlier, of reformatting a file of city and country names.

```
 1   my %table;

 2   while (<>) {
 3     chomp;
 4     my ($city, $country) = split /, /;
 5     $table{$country} = [] unless exists $table{$country};
 6     push @{$table{$country}}, $city;
 7   }

 8   for my $country (sort keys %table) {
 9     print "$country: ";
10     my @cities = @{$table{$country}};
11     print join ', ', sort @cities;
12     print ".\n";
13   }
```

The program has two pieces: Lines 2–7 read the input and build a data structure, and lines 8–13 analyze the data and print out the report. We're going to have a hash, `%table`, whose keys are country names, and whose values are references to arrays of city names. The data structure will look like this:



We'll look at output first. Supposing we already have this structure, how do we print it out?

```
 8   for my $country (sort keys %table) {
 9     print "$country: ";
10     my @cities = @{$table{$country}};
11     print join ', ', sort @cities;
12     print ".\n";
13   }
```

`%table` is an ordinary hash, and we get a list of keys from it, sort the keys, and loop over the keys as

usual. The only use of references is in line 10. `$table{$country}` looks up the key `$country` in the hash and gets the value, which is a reference to an array of cities in that country. **Use Rule 1** says that we can recover the array by saying `@{$table{$country}}`. Line 10 is just like

```
@cities = @array;
```

except that the name `array` has been replaced by the reference `{$table{$country}}`. The `@` tells Perl to get the entire array. Having gotten the list of cities, we sort it, join it, and print it out as usual.

Lines 2–7 are responsible for building the structure in the first place. Here they are again:

```
2   while (<>) {
3     chomp;
4     my ($city, $country) = split /, /;
5     $table{$country} = [] unless exists $table{$country};
6     push @{$table{$country}}, $city;
7   }
```

Lines 2–4 acquire a city and country name. Line 5 looks to see if the country is already present as a key in the hash. If it's not, the program uses the `[]` notation (**Make Rule 2**) to manufacture a new, empty anonymous array of cities, and installs a reference to it into the hash under the appropriate key.

Line 6 installs the city name into the appropriate array. `$table{$country}` now holds a reference to the array of cities seen in that country so far. Line 6 is exactly like

```
push @array, $city;
```

except that the name `array` has been replaced by the reference `{$table{$country}}`. The `push` adds a city name to the end of the referred-to array.

There's one fine point I skipped. Line 5 is unnecessary, and we can get rid of it.

```
2   while (<>) {
3     chomp;
4     my ($city, $country) = split /, /;
5   ####  $table{$country} = [] unless exists $table{$country};
6     push @{$table{$country}}, $city;
7   }
```

If there's already an entry in `%table` for the current `$country`, then nothing is different. Line 6 will locate the value in `$table{$country}`, which is a reference to an array, and push `$city` into the array. But what does it do when `$country` holds a key, say `Greece`, that is not yet in `%table`?

This is Perl, so it does the exact right thing. It sees that you want to push `Athens` onto an array that doesn't exist, so it helpfully makes a new, empty, anonymous array for you, installs it into `%table`, and then pushes `Athens` onto it. This is called *autovivification*––bringing things to life automatically. Perl saw that the key wasn't in the hash, so it created a new hash entry automatically. Perl saw that you wanted to use the hash value as an array, so it created a new empty array and installed a reference to it in the hash automatically. And as usual, Perl made the array one element longer to hold the new city name.

The Rest

I promised to give you 90% of the benefit with 10% of the details, and that means I left out 90% of the details. Now that you have an overview of the important parts, it should be easier to read the perlref manual page, which discusses 100% of the details.

Some of the highlights of perlref:

- You can make references to anything, including scalars, functions, and other references.
- In **Use Rule 1**, you can omit the curly brackets whenever the thing inside them is an atomic scalar variable like `$aref`. For example, `@$aref` is the same as `@{$aref}`, and `$$aref[1]` is the same as `${$aref}[1]`. If you're just starting out, you may want to adopt the habit of always including the curly brackets.
- This doesn't copy the underlying array:

  ```
  $aref2 = $aref1;
  ```

  You get two references to the same array. If you modify `$aref1->[23]` and then look at `$aref2->[23]` you'll see the change.

To copy the array, use

```
$aref2 = [@{$aref1}];
```

This uses `[...]` notation to create a new anonymous array, and `$aref2` is assigned a reference to the new array. The new array is initialized with the contents of the array referred to by `$aref1`.

Similarly, to copy an anonymous hash, you can use

```
$href2 = {%{$href1}};
```

- To see if a variable contains a reference, use the `ref` function. It returns true if its argument is a reference. Actually it's a little better than that: It returns `HASH` for hash references and `ARRAY` for array references.
- If you try to use a reference like a string, you get strings like

  ```
  ARRAY(0x80f5dec)   or   HASH(0x826afc0)
  ```

  If you ever see a string that looks like this, you'll know you printed out a reference by mistake.

  A side effect of this representation is that you can use `eq` to see if two references refer to the same thing. (But you should usually use == instead because it's much faster.)
- You can use a string as if it were a reference. If you use the string `"foo"` as an array reference, it's taken to be a reference to the array `@foo`. This is called a *symbolic reference*. The declaration `use strict 'refs'` disables this feature, which can cause all sorts of trouble if you use it by accident.

You might prefer to go on to perllol instead of perlref; it discusses lists of lists and multidimensional arrays in detail. After that, you should move on to perldsc; it's a Data Structure Cookbook that shows recipes for using and printing out arrays of hashes, hashes of arrays, and other kinds of data.

Summary

Everyone needs compound data structures, and in Perl the way you get them is with references. There are four important rules for managing references: Two for making references and two for using them. Once you know these rules you can do most of the important things you need to do with references.

Credits

Author: Mark Jason Dominus, Plover Systems (`mjd-perl-ref+@plover.com`)

This article originally appeared in *The Perl Journal* (<http://www.tpj.com/>) volume 3, #2. Reprinted with permission.

The original title was *Understand References Today*.

Distribution Conditions

NAME

perldsc – Perl Data Structures Cookbook

DESCRIPTION

Perl lets us have complex data structures. You can write something like this and all of a sudden, you'd have an array with three dimensions!

```
for my $x (1 .. 10) {
    for my $y (1 .. 10) {
        for my $z (1 .. 10) {
            $AoA[$x][$y][$z] =
                $x ** $y + $z;
        }
    }
}
```

Alas, however simple this may appear, underneath it's a much more elaborate construct than meets the eye!

How do you print it out? Why can't you say just `print @AoA`? How do you sort it? How can you pass it to a function or get one of these back from a function? Is it an object? Can you save it to disk to read back later? How do you access whole rows or columns of that matrix? Do all the values have to be numeric?

As you see, it's quite easy to become confused. While some small portion of the blame for this can be attributed to the reference-based implementation, it's really more due to a lack of existing documentation with examples designed for the beginner.

This document is meant to be a detailed but understandable treatment of the many different sorts of data structures you might want to develop. It should also serve as a cookbook of examples. That way, when you need to create one of these complex data structures, you can just pinch, pilfer, or purloin a drop-in example from here.

Let's look at each of these possible constructs in detail. There are separate sections on each of the following:

- arrays of arrays
- hashes of arrays
- arrays of hashes
- hashes of hashes
- more elaborate constructs

But for now, let's look at general issues common to all these types of data structures.

REFERENCES

The most important thing to understand about all data structures in Perl — including multidimensional arrays — is that even though they might appear otherwise, Perl `@ARRAY`s and `%HASH`es are all internally one-dimensional. They can hold only scalar values (meaning a string, number, or a reference). They cannot directly contain other arrays or hashes, but instead contain *references* to other arrays or hashes.

You can't use a reference to an array or hash in quite the same way that you would a real array or hash. For C or C++ programmers unused to distinguishing between arrays and pointers to the same, this can be confusing. If so, just think of it as the difference between a structure and a pointer to a structure.

You can (and should) read more about references in perlref. Briefly, references are rather like pointers that know what they point to. (Objects are also a kind of reference, but we won't be needing them right away — if ever.) This means that when you have something which looks to you like an access to a two-or-more-dimensional array and/or hash, what's really going on is that the base type is merely a one-dimensional entity that contains references to the next level. It's just that you can *use* it as though it were a two-dimensional one. This is actually the way almost all C multidimensional arrays work as well.

```
$array[7][12]                       # array of arrays
$array[7]{string}                   # array of hashes
$hash{string}[7]                    # hash of arrays
$hash{string}{'another string'}     # hash of hashes
```

Now, because the top level contains only references, if you try to print out your array in with a simple *print()* function, you'll get something that doesn't look very nice, like this:

```
    my @AoA = ( [2, 3], [4, 5, 7], [0] );
    print $AoA[1][2];
  7
    print @AoA;
  ARRAY(0x83c38)ARRAY(0x8b194)ARRAY(0x8b1d0)
```

That's because Perl doesn't (ever) implicitly dereference your variables. If you want to get at the thing a reference is referring to, then you have to do this yourself using either prefix typing indicators, like `${$blah}`, `@{$blah}`, `@{$blah[$i]}`, or else postfix pointer arrows, like `$a->[3]`, `$h->{fred}`, or even `$ob->method()->[3]`.

COMMON MISTAKES

The two most common mistakes made in constructing something like an array of arrays is either accidentally counting the number of elements or else taking a reference to the same memory location repeatedly. Here's the case where you just get the count instead of a nested array:

```
for my $i (1..10) {
    my @array = somefunc($i);
    $AoA[$i] = @array;      # WRONG!
}
```

That's just the simple case of assigning an array to a scalar and getting its element count. If that's what you really and truly want, then you might do well to consider being a tad more explicit about it, like this:

```
for my $i (1..10) {
    my @array = somefunc($i);
    $counts[$i] = scalar @array;
}
```

Here's the case of taking a reference to the same memory location again and again:

```
# Either without strict or having an outer-scope my @array;
# declaration.

for my $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = \@array;     # WRONG!
}
```

So, what's the big problem with that? It looks right, doesn't it? After all, I just told you that you need an array of references, so by golly, you've made me one!

Unfortunately, while this is true, it's still broken. All the references in `@AoA` refer to the *very same place*, and they will therefore all hold whatever was last in `@array`! It's similar to the problem demonstrated in the following C program:

```
#include <pwd.h>
main() {
    struct passwd *getpwnam(), *rp, *dp;
    rp = getpwnam("root");
    dp = getpwnam("daemon");

    printf("daemon name is %s\nroot name is %s\n",
            dp->pw_name, rp->pw_name);
}
```

Which will print

```
daemon name is daemon
root name is daemon
```

The problem is that both `rp` and `dp` are pointers to the same location in memory! In C, you'd have to remember to *malloc()* yourself some new memory. In Perl, you'll want to use the array constructor `[]` or the hash constructor `{}` instead. Here's the right way to do the preceding broken code fragments:

```
# Either without strict or having an outer-scope my @array;
# declaration.

for my $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = [ @array ];
}
```

The square brackets make a reference to a new array with a *copy* of what's in `@array` at the time of the assignment. This is what you want.

Note that this will produce something similar, but it's much harder to read:

```
# Either without strict or having an outer-scope my @array;
# declaration.
for my $i (1..10) {
    @array = 0 .. $i;
    @{$AoA[$i]} = @array;
}
```

Is it the same? Well, maybe so — and maybe not. The subtle difference is that when you assign something in square brackets, you know for sure it's always a brand new reference with a new *copy* of the data. Something else could be going on in this new case with the `@{$AoA[$i]}` dereference on the left-hand-side of the assignment. It all depends on whether `$AoA[$i]` had been undefined to start with, or whether it already contained a reference. If you had already populated `@AoA` with references, as in

```
$AoA[3] = \@another_array;
```

Then the assignment with the indirection on the left-hand-side would use the existing reference that was already there:

```
@{$AoA[3]} = @array;
```

Of course, this *would* have the "interesting" effect of clobbering `@another_array`. (Have you ever noticed how when a programmer says something is "interesting", that rather than meaning "intriguing", they're disturbingly more apt to mean that it's "annoying", "difficult", or both? :–)

So just remember always to use the array or hash constructors with `[]` or `{}`, and you'll be fine, although it's not always optimally efficient.

Surprisingly, the following dangerous-looking construct will actually work out fine:

```
for my $i (1..10) {
    my @array = somefunc($i);
    $AoA[$i] = \@array;
}
```

That's because *my()* is more of a run-time statement than it is a compile-time declaration *per se*. This means that the *my()* variable is remade afresh each time through the loop. So even though it *looks* as though you stored the same variable reference each time, you actually did not! This is a subtle distinction that can produce more efficient code at the risk of misleading all but the most experienced of programmers. So I usually advise against teaching it to beginners. In fact, except for passing arguments to functions, I seldom like to see the gimme-a-reference operator (backslash) used much at all in code. Instead, I advise beginners that they (and most of the rest of us) should try to use the much more easily understood constructors `[]` and `{}` instead of relying upon lexical (or dynamic) scoping and hidden reference-counting to do the right thing behind the scenes.

In summary:

```
$AoA[$i] = [ @array ];     # usually best
$AoA[$i] = \@array;        # perilous; just how my() was that array?
@{ $AoA[$i] } = @array;    # way too tricky for most programmers
```

CAVEAT ON PRECEDENCE

Speaking of things like `@{$AoA[$i]}`, the following are actually the same thing:

```
$aref->[2][2]       # clear
$$aref[2][2]        # confusing
```

That's because Perl's precedence rules on its five prefix dereferencers (which look like someone swearing: `$ @ * % &`) make them bind more tightly than the postfix subscripting brackets or braces! This will no doubt come as a great shock to the C or C++ programmer, who is quite accustomed to using `*a[i]` to mean what's pointed to by the *i'th* element of `a`. That is, they first take the subscript, and only then dereference the thing at that subscript. That's fine in C, but this isn't C.

The seemingly equivalent construct in Perl, `$$aref[$i]` first does the deref of `$aref`, making it take `$aref` as a reference to an array, and then dereference that, and finally tell you the *i'th* value of the array pointed to by `$AoA`. If you wanted the C notion, you'd have to write `${$AoA[$i]}` to force the `$AoA[$i]` to get evaluated first before the leading `$` dereferencer.

WHY YOU SHOULD ALWAYS `use strict`

If this is starting to sound scarier than it's worth, relax. Perl has some features to help you avoid its most common pitfalls. The best way to avoid getting confused is to start every program like this:

```
#!/usr/bin/perl -w
use strict;
```

This way, you'll be forced to declare all your variables with *my()* and also disallow accidental "symbolic dereferencing". Therefore if you'd done this:

```
my $aref = [
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $aref[2][2];
```

The compiler would immediately flag that as an error *at compile time*, because you were accidentally accessing `@aref`, an undeclared variable, and it would thereby remind you to write instead:

```
print $aref->[2][2]
```

DEBUGGING

You can use the debugger's `x` command to dump out complex data structures. For example, given the assignment to `$AoA` above, here's the debugger output:

```
DB<1> x $AoA
$AoA = ARRAY(0x13b5a0)
   0  ARRAY(0x1f0a24)
      0  'fred'
      1  'barney'
      2  'pebbles'
      3  'bambam'
      4  'dino'
   1  ARRAY(0x13b558)
      0  'homer'
      1  'bart'
      2  'marge'
      3  'maggie'
   2  ARRAY(0x13b540)
      0  'george'
      1  'jane'
      2  'elroy'
      3  'judy'
```

CODE EXAMPLES

Presented with little comment (these will get their own manpages someday) here are short code examples illustrating access of various types of data structures.

ARRAYS OF ARRAYS

Declaration of an ARRAY OF ARRAYS

```
@AoA = (
       [ "fred", "barney" ],
       [ "george", "jane", "elroy" ],
       [ "homer", "marge", "bart" ],
     );
```

Generation of an ARRAY OF ARRAYS

```
# reading from file
while ( <> ) {
    push @AoA, [ split ];
}

# calling a function
for $i ( 1 .. 10 ) {
    $AoA[$i] = [ somefunc($i) ];
}

# using temp vars
for $i ( 1 .. 10 ) {
    @tmp = somefunc($i);
    $AoA[$i] = [ @tmp ];
}

# add to an existing row
push @{ $AoA[0] }, "wilma", "betty";
```

Access and Printing of an ARRAY OF ARRAYS

```
# one element
$AoA[0][0] = "Fred";

# another element
$AoA[1][1] =~ s/(\w)/\u$1/;

# print the whole thing with refs
for $aref ( @AoA ) {
    print "\t [ @$aref ],\n";
}

# print the whole thing with indices
for $i ( 0 .. $#AoA ) {
    print "\t [ @{$AoA[$i]} ],\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. $#{ $AoA[$i] } ) {
        print "elt $i $j is $AoA[$i][$j]\n";
    }
}
```

HASHES OF ARRAYS

Declaration of a HASH OF ARRAYS

```
%HoA = (
       flintstones        => [ "fred", "barney" ],
       jetsons            => [ "george", "jane", "elroy" ],
       simpsons           => [ "homer", "marge", "bart" ],
     );
```

Generation of a HASH OF ARRAYS

```
# reading from file
# flintstones: fred barney wilma dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $HoA{$1} = [ split ];
}

# reading from file; more temps
# flintstones: fred barney wilma dino
while ( $line = <> ) {
    ($who, $rest) = split /:\s*/, $line, 2;
    @fields = split ' ', $rest;
    $HoA{$who} = [ @fields ];
}

# calling a function that returns a list
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoA{$group} = [ get_family($group) ];
}

# likewise, but using temps
for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoA{$group} = [ @members ];
}

# append new members to an existing family
push @{ $HoA{"flintstones"} }, "wilma", "betty";
```

Access and Printing of a HASH OF ARRAYS

```
# one element
$HoA{flintstones}[0] = "Fred";

# another element
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}

# print the whole thing with indices
foreach $family ( keys %HoA ) {
    print "family: ";
    foreach $i ( 0 .. $#{ $HoA{$family} } ) {
        print " $i = $HoA{$family}[$i]";
    }
    print "\n";
}

# print the whole thing sorted by number of members
foreach $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}

# print the whole thing sorted by number of members and name
foreach $family ( sort {
                            @{$HoA{$b}} <=> @{$HoA{$a}}
```

```
                                            ||
                                          $a cmp $b
            } keys %HoA )
    {
        print "$family: ", join(", ", sort @{ $HoA{$family} }), "\n";
    }
```

ARRAYS OF HASHES

Declaration of an ARRAY OF HASHES

```
    @AoH = (
           {
               Lead     => "fred",
               Friend   => "barney",
           },
           {
               Lead     => "george",
               Wife     => "jane",
               Son      => "elroy",
           },
           {
               Lead     => "homer",
               Wife     => "marge",
               Son      => "bart",
           }
     );
```

Generation of an ARRAY OF HASHES

```
    # reading from file
    # format: LEAD=fred FRIEND=barney
    while ( <> ) {
        $rec = {};
        for $field ( split ) {
            ($key, $value) = split /=/, $field;
            $rec->{$key} = $value;
        }
        push @AoH, $rec;
    }


    # reading from file
    # format: LEAD=fred FRIEND=barney
    # no temp
    while ( <> ) {
        push @AoH, { split /[\s+=]/ };
    }

    # calling a function  that returns a key/value pair list, like
    # "lead","fred","daughter","pebbles"
    while ( %fields = getnextpairset() ) {
        push @AoH, { %fields };
    }

    # likewise, but using no temp vars
    while (<>) {
        push @AoH, { parsepairs($_) };
    }

    # add key/value to an element
    $AoH[0]{pet} = "dino";
    $AoH[2]{pet} = "santa's little helper";
```

Access and Printing of an ARRAY OF HASHES

```
# one element
$AoH[0]{lead} = "fred";

# another element
$AoH[1]{lead} =~ s/(\w)/\u$1/;

# print the whole thing with refs
for $href ( @AoH ) {
    print "{ ";
    for $role ( keys %$href ) {
        print "$role=$href->{$role} ";
    }
    print "}\n";
}

# print the whole thing with indices
for $i ( 0 .. $#AoH ) {
    print "$i is { ";
    for $role ( keys %{ $AoH[$i] } ) {
        print "$role=$AoH[$i]{$role} ";
    }
    print "}\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#AoH ) {
    for $role ( keys %{ $AoH[$i] } ) {
        print "elt $i $role is $AoH[$i]{$role}\n";
    }
}
```

HASHES OF HASHES

Declaration of a HASH OF HASHES

```
%HoH = (
       flintstones => {
               lead      => "fred",
               pal       => "barney",
       },
       jetsons     => {
               lead      => "george",
               wife      => "jane",
               "his boy" => "elroy",
       },
       simpsons    => {
               lead      => "homer",
               wife      => "marge",
               kid       => "bart",
       },
);
```

Generation of a HASH OF HASHES

```
# reading from file
# flintstones: lead=fred pal=barney wife=wilma pet=dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $HoH{$who}{$key} = $value;
    }


# reading from file; more temps
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    $rec = {};
    $HoH{$who} = $rec;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
}

# calling a function  that returns a key,value hash
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoH{$group} = { get_family($group) };
}

# likewise, but using temps
for $group ( "simpsons", "jetsons", "flintstones" ) {
    %members = get_family($group);
    $HoH{$group} = { %members };
}

# append new members to an existing family
%new_folks = (
    wife => "wilma",
    pet  => "dino",
);

for $what (keys %new_folks) {
    $HoH{flintstones}{$what} = $new_folks{$what};
}
```

Access and Printing of a HASH OF HASHES

```
# one element
$HoH{flintstones}{wife} = "wilma";

# another element
$HoH{simpsons}{lead} =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoH ) {
    print "$family: { ";
    for $role ( keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}
```

```
# print the whole thing  somewhat sorted
foreach $family ( sort keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}


# print the whole thing sorted by number of members
foreach $family ( sort { keys %{$HoH{$b}} <=> keys %{$HoH{$a}} }
                                                        keys %HoH )
{
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# establish a sort order (rank) for each role
$i = 0;
for ( qw(lead wife son daughter pal pet) ) { $rank{$_} = ++$i }

# now print the whole thing sorted by number of members
foreach $family ( sort { keys %{ $HoH{$b} } <=> keys %{ $HoH{$a} } }
                                                        keys %HoH )
{
    print "$family: { ";
    # and print these according to rank order
    for $role ( sort { $rank{$a} <=> $rank{$b} }
                                            keys %{ $HoH{$family} } )
    {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}
```

MORE ELABORATE RECORDS

Declaration of MORE ELABORATE RECORDS

Here's a sample showing how to create and use a record whose fields are of many different sorts:

```
$rec = {
    TEXT      => $string,
    SEQUENCE  => [ @old_values ],
    LOOKUP    => { %some_table },
    THATCODE  => \&some_function,
    THISCODE  => sub { $_[0] ** $_[1] },
    HANDLE    => \*STDOUT,
};

print $rec->{TEXT};

print $rec->{SEQUENCE}[0];
$last = pop @ { $rec->{SEQUENCE} };

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} };
```

```
$answer = $rec->{THATCODE}->($arg);
$answer = $rec->{THISCODE}->($arg1, $arg2);

# careful of extra block braces on fh ref
print { $rec->{HANDLE} } "a string\n";

use FileHandle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print(" a string\n");
```

Declaration of a HASH OF COMPLEX RECORDS

```
%TV = (
   flintstones => {
       series   => "flintstones",
       nights   => [ qw(monday thursday friday) ],
       members  => [
           { name => "fred",    role => "lead", age  => 36, },
           { name => "wilma",   role => "wife", age  => 31, },
           { name => "pebbles", role => "kid",  age  =>  4, },
       ],
   },

   jetsons     => {
       series   => "jetsons",
       nights   => [ qw(wednesday saturday) ],
       members  => [
           { name => "george",  role => "lead", age  => 41, },
           { name => "jane",    role => "wife", age  => 39, },
           { name => "elroy",   role => "kid",  age  =>  9, },
       ],
    },

   simpsons    => {
       series   => "simpsons",
       nights   => [ qw(monday) ],
       members  => [
           { name => "homer", role => "lead", age  => 34, },
           { name => "marge", role => "wife", age => 37, },
           { name => "bart",  role => "kid",  age  =>  11, },
       ],
    },
 );
```

Generation of a HASH OF COMPLEX RECORDS

```
# reading from file
# this is most easily done by having the file itself be
# in the raw data format as shown above.  perl is happy
# to parse complex data structures if declared as data, so
# sometimes it's easiest to do that

# here's a piece by piece build up
$rec = {};
$rec->{series} = "flintstones";
$rec->{nights} = [ find_days() ];

@members = ();
# assume this file in field=value syntax
while (<>) {
    %fields = split /[\s=]+/;
```

```
    push @members, { %fields };
}
$rec->{members} = [ @members ];

# now remember the whole thing
$TV{ $rec->{series} } = $rec;

############################################################
# now, you might want to make interesting extra fields that
# include pointers back into the same data structure so if
# change one piece, it changes everywhere, like for example
# if you wanted a {kids} field that was a reference
# to an array of the kids' records without having duplicate
# records and thus update problems.
############################################################
foreach $family (keys %TV) {
    $rec = $TV{$family}; # temp pointer
    @kids = ();
    for $person ( @{ $rec->{members} } ) {
        if ($person->{role} =~ /kid|son|daughter/) {
            push @kids, $person;
        }
    }
    # REMEMBER: $rec and $TV{$family} point to same data!!
    $rec->{kids} = [ @kids ];
}

# you copied the array, but the array itself contains pointers
# to uncopied objects. this means that if you make bart get
# older via

$TV{simpsons}{kids}[0]{age}++;

# then this would also change in
print $TV{simpsons}{members}[2]{age};

# because $TV{simpsons}{kids}[0] and $TV{simpsons}{members}[2]
# both point to the same underlying anonymous hash table

# print the whole thing
foreach $family ( keys %TV ) {
    print "the $family";
    print " is on during @{ $TV{$family}{nights} }\n";
    print "its members are:\n";
    for $who ( @{ $TV{$family}{members} } ) {
        print " $who->{name} ($who->{role}), age $who->{age}\n";
    }
    print "it turns out that $TV{$family}{lead} has ";
    print scalar ( @{ $TV{$family}{kids} } ), " kids named ";
    print join (", ", map { $_->{name} } @{ $TV{$family}{kids} } );
    print "\n";
}
```

Database Ties

You cannot easily tie a multilevel data structure (such as a hash of hashes) to a dbm file. The first problem is that all but GDBM and Berkeley DB have size limitations, but beyond that, you also have problems with how references are to be represented on disk. One experimental module that does partially attempt to address this need is the MLDBM module. Check your nearest CPAN site as described in perlmodlib for source code to MLDBM.

SEE ALSO

perlref, perllol, perldata, perlobj

AUTHOR

Tom Christiansen <*tchrist@perl.com*>

NAME

perlrequick – Perl regular expressions quick start

DESCRIPTION

This page covers the very basics of understanding, creating and using regular expressions ('regexes') in Perl.

The Guide

This page assumes you already know things, like what a "pattern" is, and the basic syntax of using them. If you don't, see perlretut.

Simple word matching

The simplest regex is simply a word, or more generally, a string of characters. A regex consisting of a word matches any string that contains that word:

```
"Hello World" =~ /World/;  # matches
```

In this statement, `World` is a regex and the `//` enclosing `/World/` tells Perl to search a string for a match. The operator `=~` associates the string with the regex match and produces a true value if the regex matched, or false if the regex did not match. In our case, `World` matches the second word in `"Hello World"`, so the expression is true. This idea has several variations.

Expressions like this are useful in conditionals:

```
print "It matches\n" if "Hello World" =~ /World/;
```

The sense of the match can be reversed by using `!~` operator:

```
print "It doesn't match\n" if "Hello World" !~ /World/;
```

The literal string in the regex can be replaced by a variable:

```
$greeting = "World";
print "It matches\n" if "Hello World" =~ /$greeting/;
```

If you're matching against `$_`, the `$_ =~` part can be omitted:

```
$_ = "Hello World";
print "It matches\n" if /World/;
```

Finally, the `//` default delimiters for a match can be changed to arbitrary delimiters by putting an `'m'` out front:

```
"Hello World" =~ m!World!;   # matches, delimited by '!'
"Hello World" =~ m{World};   # matches, note the matching '{}'
"/usr/bin/perl" =~ m"/perl"; # matches after '/usr/bin',
                             # '/' becomes an ordinary char
```

Regexes must match a part of the string *exactly* in order for the statement to be true:

```
"Hello World" =~ /world/;  # doesn't match, case sensitive
"Hello World" =~ /o W/;    # matches, ' ' is an ordinary char
"Hello World" =~ /World /; # doesn't match, no ' ' at end
```

Perl will always match at the earliest possible point in the string:

```
"Hello World" =~ /o/;       # matches 'o' in 'Hello'
"That hat is red" =~ /hat/; # matches 'hat' in 'That'
```

Not all characters can be used 'as is' in a match. Some characters, called **metacharacters**, are reserved for use in regex notation. The metacharacters are

```
{}[]()^$.|*+?\
```

A metacharacter can be matched by putting a backslash before it:

```
"2+2=4" =~ /2+2/;     # doesn't match, + is a metacharacter
"2+2=4" =~ /2\+2/;    # matches, \+ is treated like an ordinary +
'C:\WIN32' =~ /C:\\WIN/;                      # matches
"/usr/bin/perl" =~ /\/usr\/bin\/perl/;  # matches
```

In the last regex, the forward slash `'/'` is also backslashed, because it is used to delimit the regex.

Non-printable ASCII characters are represented by **escape sequences**. Common examples are `\t` for a tab,

`\n` for a newline, and `\r` for a carriage return. Arbitrary bytes are represented by octal escape sequences, e.g., `\033`, or hexadecimal escape sequences, e.g., `\x1B`:

```
"1000\t2000" =~ m(0\t2)  # matches
"cat" =~ /\143\x61\x74/  # matches in ASCII, but
                         # a weird way to spell cat
```

Regexes are treated mostly as double-quoted strings, so variable substitution works:

```
$foo = 'house';
'cathouse' =~ /cat$foo/;   # matches
'housecat' =~ /${foo}cat/; # matches
```

With all of the regexes above, if the regex matched anywhere in the string, it was considered a match. To specify *where* it should match, we would use the **anchor** metacharacters `^` and `$`. The anchor `^` means match at the beginning of the string and the anchor `$` means match at the end of the string, or before a newline at the end of the string. Some examples:

```
"housekeeper" =~ /keeper/;          # matches
"housekeeper" =~ /^keeper/;         # doesn't match
"housekeeper" =~ /keeper$/;         # matches
"housekeeper\n" =~ /keeper$/;       # matches
"housekeeper" =~ /^housekeeper$/;  # matches
```

Using character classes

A **character class** allows a set of possible characters, rather than just a single character, to match at a particular point in a regex. Character classes are denoted by brackets `[...]`, with the set of characters to be possibly matched inside. Here are some examples:

```
/cat/;             # matches 'cat'
/[bcr]at/;         # matches 'bat', 'cat', or 'rat'
"abc" =~ /[cab]/; # matches 'a'
```

In the last statement, even though `'c'` is the first character in the class, the earliest point at which the regex can match is `'a'`.

```
/[yY][eE][sS]/; # match 'yes' in a case-insensitive way
                # 'yes', 'Yes', 'YES', etc.
/yes/i;         # also match 'yes' in a case-insensitive way
```

The last example shows a match with an `'i'` **modifier**, which makes the match case-insensitive.

Character classes also have ordinary and special characters, but the sets of ordinary and special characters inside a character class are different than those outside a character class. The special characters for a character class are `-]\^$` and are matched using an escape:

```
/[\]c]def/; # matches ']def' or 'cdef'
$x = 'bcr';
/[$x]at/;   # matches 'bat, 'cat', or 'rat'
/[\$x]at/;  # matches '$at' or 'xat'
/[\\$x]at/; # matches '\at', 'bat, 'cat', or 'rat'
```

The special character `'-'` acts as a range operator within character classes, so that the unwieldy `[0123456789]` and `[abc...xyz]` become the svelte `[0-9]` and `[a-z]`:

```
/item[0-9]/;  # matches 'item0' or ... or 'item9'
/[0-9a-fA-F]/;  # matches a hexadecimal digit
```

If `'-'` is the first or last character in a character class, it is treated as an ordinary character.

The special character `^` in the first position of a character class denotes a **negated character class**, which matches any character but those in the brackets. Both `[...]` and `[^...]` must match a character, or the match fails. Then

```
/[^a]at/;  # doesn't match 'aat' or 'at', but matches
           # all other 'bat', 'cat, '0at', '%at', etc.
/[^0-9]/;  # matches a non-numeric character
/[a^]at/;  # matches 'aat' or '^at'; here '^' is ordinary
```

Perl has several abbreviations for common character classes. (These definitions are those that Perl uses in ASCII-safe mode with the `/a` modifier. Otherwise they could match many more non-ASCII Unicode characters as well. See "Backslash sequences" in perlrecharclass for details.)

- \d is a digit and represents

  ```
  [0-9]
  ```

- \s is a whitespace character and represents

  ```
  [\ \t\r\n\f]
  ```

- \w is a word character (alphanumeric or _) and represents

  ```
  [0-9a-zA-Z_]
  ```

- \D is a negated \d; it represents any character but a digit

  ```
  [^0-9]
  ```

- \S is a negated \s; it represents any non-whitespace character

  ```
  [^\s]
  ```

- \W is a negated \w; it represents any non-word character

  ```
  [^\w]
  ```

- The period '.' matches any character but "\n"

The `\d\s\w\D\S\W` abbreviations can be used both inside and outside of character classes. Here are some in use:

```
/\d\d:\d\d:\d\d/; # matches a hh:mm:ss time format
/[\d\s]/;         # matches any digit or whitespace character
/\w\W\w/;         # matches a word char, followed by a
                  # non-word char, followed by a word char
/..rt/;           # matches any two chars, followed by 'rt'
/end\./;          # matches 'end.'
/end[.]/;         # same thing, matches 'end.'
```

The **word anchor** `\b` matches a boundary between a word character and a non-word character `\w\W` or `\W\w`:

```
$x = "Housecat catenates house and cat";
$x =~ /\bcat/;  # matches cat in 'catenates'
$x =~ /cat\b/;  # matches cat in 'housecat'
$x =~ /\bcat\b/;  # matches 'cat' at end of string
```

In the last example, the end of the string is considered a word boundary.

For natural language processing (so that, for example, apostrophes are included in words), use instead `\b{wb}`

```
"don't" =~ / .+? \b{wb} /x;  # matches the whole string
```

Matching this or that

We can match different character strings with the **alternation** metacharacter `'|'`. To match `dog` or `cat`, we form the regex `dog|cat`. As before, Perl will try to match the regex at the earliest possible point in the string. At each character position, Perl will first try to match the first alternative, `dog`. If `dog` doesn't match, Perl will then try the next alternative, `cat`. If `cat` doesn't match either, then the match fails and Perl moves to the next position in the string. Some examples:

```
"cats and dogs" =~ /cat|dog|bird/;  # matches "cat"
"cats and dogs" =~ /dog|cat|bird/;  # matches "cat"
```

Even though `dog` is the first alternative in the second regex, `cat` is able to match earlier in the string.

```
"cats"          =~ /c|ca|cat|cats/; # matches "c"
"cats"          =~ /cats|cat|ca|c/; # matches "cats"
```

At a given character position, the first alternative that allows the regex match to succeed will be the one that matches. Here, all the alternatives match at the first string position, so the first matches.

Grouping things and hierarchical matching

The **grouping** metacharacters `()` allow a part of a regex to be treated as a single unit. Parts of a regex are grouped by enclosing them in parentheses. The regex `house(cat|keeper)` means match `house` followed by either `cat` or `keeper`. Some more examples are

```
/(a|b)b/;    # matches 'ab' or 'bb'
/(^a|b)c/;   # matches 'ac' at start of string or 'bc' anywhere

/house(cat|)/;  # matches either 'housecat' or 'house'
/house(cat(s|)|)/;  # matches either 'housecats' or 'housecat' or
                    # 'house'.  Note groups can be nested.

"20" =~ /(19|20|)\d\d/;  # matches the null alternative '()\d\d',
                         # because '20\d\d' can't match
```

Extracting matches

The grouping metacharacters `()` also allow the extraction of the parts of a string that matched. For each grouping, the part that matched inside goes into the special variables `$1`, `$2`, etc. They can be used just as ordinary variables:

```
# extract hours, minutes, seconds
$time =~ /(\d\d):(\d\d):(\d\d)/;  # match hh:mm:ss format
$hours = $1;
$minutes = $2;
$seconds = $3;
```

In list context, a match `/regex/` with groupings will return the list of matched values `($1,$2,...)`. So we could rewrite it as

```
($hours, $minutes, $second) = ($time =~ /(\d\d):(\d\d):(\d\d)/);
```

If the groupings in a regex are nested, `$1` gets the group with the leftmost opening parenthesis, `$2` the next opening parenthesis, etc. For example, here is a complex regex and the matching variables indicated below it:

```
/(ab(cd|ef)((gi)|j))/;
 1  2      34
```

Associated with the matching variables `$1`, `$2`, ... are the **backreferences** `\g1`, `\g2`, ... Backreferences are matching variables that can be used *inside* a regex:

```
/(\w\w\w)\s\g1/; # find sequences like 'the the' in string
```

`$1`, `$2`, ... should only be used outside of a regex, and `\g1`, `\g2`, ... only inside a regex.

Matching repetitions

The **quantifier** metacharacters `?`, `*`, `+`, and `{}` allow us to determine the number of repeats of a portion of a regex we consider to be a match. Quantifiers are put immediately after the character, character class, or grouping that we want to specify. They have the following meanings:

- `a?` = match 'a' 1 or 0 times
- `a*` = match 'a' 0 or more times, i.e., any number of times
- `a+` = match 'a' 1 or more times, i.e., at least once
- `a{n,m}` = match at least `n` times, but not more than `m` times.
- `a{n,}` = match at least `n` or more times
- `a{n}` = match exactly `n` times

Here are some examples:

```
/[a-z]+\s+\d*/;  # match a lowercase word, at least some space, and
                 # any number of digits
/(\w+)\s+\g1/;    # match doubled words of arbitrary length
$year =~ /^\d{2,4}$/;  # make sure year is at least 2 but not more
                       # than 4 digits
$year =~ /^\d{4}$|^\d{2}$/; # better match; throw out 3 digit dates
```

These quantifiers will try to match as much of the string as possible, while still allowing the regex to match. So we have

```
$x = 'the cat in the hat';
$x =~ /^(.*)(at)(.*)$/; # matches,
                        # $1 = 'the cat in the h'
                        # $2 = 'at'
                        # $3 = ''   (0 matches)
```

The first quantifier `.*` grabs as much of the string as possible while still having the regex match. The second quantifier `.*` has no string left to it, so it matches 0 times.

More matching

There are a few more things you might want to know about matching operators. The global modifier `/g` allows the matching operator to match within a string as many times as possible. In scalar context, successive matches against a string will have `/g` jump from match to match, keeping track of position in the string as it goes along. You can get or set the position with the `pos()` function. For example,

```
$x = "cat dog house"; # 3 words
while ($x =~ /(\w+)/g) {
    print "Word is $1, ends at position ", pos $x, "\n";
}
```

prints

```
Word is cat, ends at position 3
Word is dog, ends at position 7
Word is house, ends at position 13
```

A failed match or changing the target string resets the position. If you don't want the position reset after failure to match, add the `/c`, as in `/regex/gc`.

In list context, `/g` returns a list of matched groupings, or if there are no groupings, a list of matches to the whole regex. So

```
@words = ($x =~ /(\w+)/g);  # matches,
                            # $word[0] = 'cat'
                            # $word[1] = 'dog'
                            # $word[2] = 'house'
```

Search and replace

Search and replace is performed using `s/regex/replacement/modifiers`. The `replacement` is a Perl double-quoted string that replaces in the string whatever is matched with the `regex`. The operator `=~` is also used here to associate a string with `s///`. If matching against `$_`, the `$_ =~` can be dropped. If there is a match, `s///` returns the number of substitutions made; otherwise it returns false. Here are a few examples:

```
$x = "Time to feed the cat!";
$x =~ s/cat/hacker/;   # $x contains "Time to feed the hacker!"
$y = "'quoted words'";
$y =~ s/^'(.*)'$/$1/;  # strip single quotes,
                       # $y contains "quoted words"
```

With the `s///` operator, the matched variables `$1`, `$2`, etc. are immediately available for use in the replacement expression. With the global modifier, `s///g` will search and replace all occurrences of the regex in the string:

```
$x = "I batted 4 for 4";
$x =~ s/4/four/;   # $x contains "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g;  # $x contains "I batted four for four"
```

The non-destructive modifier `s///r` causes the result of the substitution to be returned instead of modifying `$_` (or whatever variable the substitute was bound to with `=~`):

```
$x = "I like dogs.";
$y = $x =~ s/dogs/cats/r;
print "$x $y\n"; # prints "I like dogs. I like cats."

$x = "Cats are great.";
print $x =~ s/Cats/Dogs/r =~ s/Dogs/Frogs/r =~
    s/Frogs/Hedgehogs/r, "\n";
# prints "Hedgehogs are great."

@foo = map { s/[a-z]/X/r } qw(a b c 1 2 3);
# @foo is now qw(X X X 1 2 3)
```

The evaluation modifier `s///e` wraps an `eval{...}` around the replacement string and the evaluated result is substituted for the matched substring. Some examples:

```
# reverse all the words in a string
$x = "the cat in the hat";
$x =~ s/(\w+)/reverse $1/ge;   # $x contains "eht tac ni eht tah"

# convert percentage to decimal
$x = "A 39% hit rate";
$x =~ s!(\d+)%!$1/100!e;       # $x contains "A 0.39 hit rate"
```

The last example shows that `s///` can use other delimiters, such as `s!!!` and `s{}{}`, and even `s{}//`. If single quotes are used `s'''`, then the regex and replacement are treated as single-quoted strings.

The split operator

`split /regex/, string` splits `string` into a list of substrings and returns that list. The regex determines the character sequence that `string` is split with respect to. For example, to split a string into words, use

```
$x = "Calvin and Hobbes";
@word = split /\s+/, $x;  # $word[0] = 'Calvin'
                          # $word[1] = 'and'
                          # $word[2] = 'Hobbes'
```

To extract a comma-delimited list of numbers, use

```
$x = "1.618,2.718,   3.142";
@const = split /,\s*/, $x;  # $const[0] = '1.618'
                            # $const[1] = '2.718'
                            # $const[2] = '3.142'
```

If the empty regex `//` is used, the string is split into individual characters. If the regex has groupings, then the list produced contains the matched substrings from the groupings as well:

```
$x = "/usr/bin";
@parts = split m!(/)!, $x;  # $parts[0] = ''
                            # $parts[1] = '/'
                            # $parts[2] = 'usr'
                            # $parts[3] = '/'
                            # $parts[4] = 'bin'
```

Since the first character of `$x` matched the regex, `split` prepended an empty initial element to the list.

`use re 'strict'`

New in v5.22, this applies stricter rules than otherwise when compiling regular expression patterns. It can find things that, while legal, may not be what you intended.

See 'strict' in re.

BUGS

None.

SEE ALSO

This is just a quick start guide. For a more in-depth tutorial on regexes, see perlretut and for the reference page, see perlre.

AUTHOR AND COPYRIGHT

Acknowledgments

The author would like to thank Mark-Jason Dominus, Tom Christiansen, Ilya Zakharevich, Brad Hughes, and Mike Giroux for all their helpful comments.

NAME

perlstyle – Perl style guide

DESCRIPTION

Each programmer will, of course, have his or her own preferences in regards to formatting, but there are some general guidelines that will make your programs easier to read, understand, and maintain.

The most important thing is to run your programs under the **-w** flag at all times. You may turn it off explicitly for particular portions of code via the `no warnings` pragma or the `$^W` variable if you must. You should also always run under `use strict` or know the reason why not. The `use sigtrap` and even `use diagnostics` pragmas may also prove useful.

Regarding aesthetics of code lay out, about the only thing Larry cares strongly about is that the closing curly bracket of a multi-line BLOCK should line up with the keyword that started the construct. Beyond that, he has other preferences that aren't so strong:

- 4-column indent.
- Opening curly on same line as keyword, if possible, otherwise line up.
- Space before the opening curly of a multi-line BLOCK.
- One-line BLOCK may be put on one line, including curlies.
- No space before the semicolon.
- Semicolon omitted in "short" one-line BLOCK.
- Space around most operators.
- Space around a "complex" subscript (inside brackets).
- Blank lines between chunks that do different things.
- Uncuddled elses.
- No space between function name and its opening parenthesis.
- Space after each comma.
- Long lines broken after an operator (except `and` and `or`).
- Space after last parenthesis matching on current line.
- Line up corresponding items vertically.
- Omit redundant punctuation as long as clarity doesn't suffer.

Larry has his reasons for each of these things, but he doesn't claim that everyone else's mind works the same as his does.

Here are some other more substantive style issues to think about:

- Just because you *CAN* do something a particular way doesn't mean that you *SHOULD* do it that way. Perl is designed to give you several ways to do anything, so consider picking the most readable one. For instance

  ```
  open(FOO,$foo) || die "Can't open $foo: $!";
  ```

  is better than

  ```
  die "Can't open $foo: $!" unless open(FOO,$foo);
  ```

  because the second way hides the main point of the statement in a modifier. On the other hand

  ```
  print "Starting analysis\n" if $verbose;
  ```

  is better than

  ```
  $verbose && print "Starting analysis\n";
  ```

  because the main point isn't whether the user typed **-v** or not.

  Similarly, just because an operator lets you assume default arguments doesn't mean that you have to make use of the defaults. The defaults are there for lazy systems programmers writing one-shot programs. If you want your program to be readable, consider supplying the argument.

Along the same lines, just because you *CAN* omit parentheses in many places doesn't mean that you ought to:

```
return print reverse sort num values %array;
return print(reverse(sort num (values(%array))));
```

When in doubt, parenthesize. At the very least it will let some poor schmuck bounce on the % key in **vi**.

Even if you aren't in doubt, consider the mental welfare of the person who has to maintain the code after you, and who will probably put parentheses in the wrong place.

- Don't go through silly contortions to exit a loop at the top or the bottom, when Perl provides the `last` operator so you can exit in the middle. Just ''outdent'' it a little to make it more visible:

```
LINE:
    for (;;) {
        statements;
      last LINE if $foo;
        next LINE if /^#/;
        statements;
    }
```

- Don't be afraid to use loop labels — they're there to enhance readability as well as to allow multilevel loop breaks. See the previous example.
- Avoid using `grep()` (or `map()`) or `backticks` in a void context, that is, when you just throw away their return values. Those functions all have return values, so use them. Otherwise use a `foreach()` loop or the `system()` function instead.
- For portability, when using features that may not be implemented on every machine, test the construct in an eval to see if it fails. If you know what version or patchlevel a particular feature was implemented, you can test `$]` (`$PERL_VERSION` in `English`) to see if it will be there. The `Config` module will also let you interrogate values determined by the **Configure** program when Perl was installed.
- Choose mnemonic identifiers. If you can't remember what mnemonic means, you've got a problem.
- While short identifiers like `$gotit` are probably ok, use underscores to separate words in longer identifiers. It is generally easier to read `$var_names_like_this` than `$VarNamesLikeThis`, especially for non-native speakers of English. It's also a simple rule that works consistently with `VAR_NAMES_LIKE_THIS`.

  Package names are sometimes an exception to this rule. Perl informally reserves lowercase module names for ''pragma'' modules like `integer` and `strict`. Other modules should begin with a capital letter and use mixed case, but probably without underscores due to limitations in primitive file systems' representations of module names as files that must fit into a few sparse bytes.
- You may find it helpful to use letter case to indicate the scope or nature of a variable. For example:

```
$ALL_CAPS_HERE   constants only (beware clashes with perl vars!)
$Some_Caps_Here  package-wide global/static
$no_caps_here    function scope my() or local() variables
```

  Function and method names seem to work best as all lowercase. E.g., `$obj->as_string()`.

  You can use a leading underscore to indicate that a variable or function should not be used outside the package that defined it.
- If you have a really hairy regular expression, use the `/x` or `/xx` modifiers and put in some whitespace to make it look a little less like line noise. Don't use slash as a delimiter when your regexp has slashes or backslashes.
- Use the new `and` and `or` operators to avoid having to parenthesize list operators so much, and to reduce the incidence of punctuation operators like `&&` and `||`. Call your subroutines as if they were functions or list operators to avoid excessive ampersands and parentheses.
- Use here documents instead of repeated `print()` statements.

- Line up corresponding things vertically, especially if it'd be too long to fit on one line anyway.

```
$IDX = $ST_MTIME;
$IDX = $ST_ATIME        if $opt_u;
$IDX = $ST_CTIME        if $opt_c;
$IDX = $ST_SIZE         if $opt_s;

mkdir $tmpdir, 0700 or die "can't mkdir $tmpdir: $!";
chdir($tmpdir)      or die "can't chdir $tmpdir: $!";
mkdir 'tmp',   0777 or die "can't mkdir $tmpdir/tmp: $!";
```

- Always check the return codes of system calls. Good error messages should go to `STDERR`, include which program caused the problem, what the failed system call and arguments were, and (VERY IMPORTANT) should contain the standard system error message for what went wrong. Here's a simple but sufficient example:

```
opendir(D, $dir)     or die "can't opendir $dir: $!";
```

- Line up your transliterations when it makes sense:

```
tr [abc]
   [xyz];
```

- Think about reusability. Why waste brainpower on a one-shot when you might want to do something like it again? Consider generalizing your code. Consider writing a module or object class. Consider making your code run cleanly with `use strict` and `use warnings` (or **-w**) in effect. Consider giving away your code. Consider changing your whole world view. Consider... oh, never mind.
- Try to document your code and use Pod formatting in a consistent way. Here are commonly expected conventions:
  - use `C<>` for function, variable and module names (and more generally anything that can be considered part of code, like filehandles or specific values). Note that function names are considered more readable with parentheses after their name, that is `function()`.
  - use `B<>` for commands names like **cat** or **grep**.
  - use `F<>` or `C<>` for file names. `F<>` should be the only Pod code for file names, but as most Pod formatters render it as italic, Unix and Windows paths with their slashes and backslashes may be less readable, and better rendered with `C<>`.
- Be consistent.
- Be nice.

NAME

perltrap – Perl traps for the unwary

DESCRIPTION

The biggest trap of all is forgetting to `use warnings` or use the **-w** switch; see warnings and perlrun. The second biggest trap is not making your entire program runnable under `use strict`. The third biggest trap is not reading the list of changes in this version of Perl; see perldelta.

Awk Traps

Accustomed **awk** users should take special note of the following:

- A Perl program executes only once, not once for each input line. You can do an implicit loop with `-n` or `-p`.
- The English module, loaded via

  ```
  use English;
  ```

  allows you to refer to special variables (like `$/`) with names (like `$RS`), as though they were in **awk**; see perlvar for details.
- Semicolons are required after all simple statements in Perl (except at the end of a block). Newline is not a statement delimiter.
- Curly brackets are required on `if`s and `while`s.
- Variables begin with "$", "@" or "%" in Perl.
- Arrays index from 0. Likewise string positions in *substr()* and *index()*.
- You have to decide whether your array has numeric or string indices.
- Hash values do not spring into existence upon mere reference.
- You have to decide whether you want to use string or numeric comparisons.
- Reading an input line does not split it for you. You get to split it to an array yourself. And the *split()* operator has different arguments than **awk**'s.
- The current input line is normally in `$_`, not `$0`. It generally does not have the newline stripped. ($0 is the name of the program executed.) See perlvar.
- $<*digit*> does not refer to fields — it refers to substrings matched by the last match pattern.
- The *print()* statement does not add field and record separators unless you set `$,` and `$\`. You can set `$OFS` and `$ORS` if you're using the English module.
- You must open your files before you print to them.
- The range operator is "..", not comma. The comma operator works as in C.
- The match operator is "=~", not "~". ("~" is the one's complement operator, as in C.)
- The exponentiation operator is "**", not "^". "^" is the XOR operator, as in C. (You know, one could get the feeling that **awk** is basically incompatible with C.)
- The concatenation operator is ".", not the null string. (Using the null string would render `/pat/ /pat/` unparsable, because the third slash would be interpreted as a division operator — the tokenizer is in fact slightly context sensitive for operators like "/", "?", and ">". And in fact, "." itself can be the beginning of a number.)
- The `next`, `exit`, and `continue` keywords work differently.
- The following variables work differently:

```
Awk       Perl
ARGC      scalar @ARGV (compare with $#ARGV)
ARGV[0]   $0
FILENAME  $ARGV
FNR       $. - something
FS        (whatever you like)
NF        $#Fld, or some such
NR        $.
OFMT      $#
OFS       $,
ORS       $\
RLENGTH   length($&)
RS        $/
RSTART    length($`)
SUBSEP    $;
```

- You cannot set `$RS` to a pattern, only a string.
- When in doubt, run the **awk** construct through **a2p** and see what it gives you.

C/C++ Traps

Cerebral C and C++ programmers should take note of the following:

- Curly brackets are required on `if`'s and `while`'s.
- You must use `elsif` rather than `else if`.
- The `break` and `continue` keywords from C become in Perl `last` and `next`, respectively. Unlike in C, these do *not* work within a `do { } while` construct. See "Loop Control" in perlsyn.
- The switch statement is called `given`/`when` and only available in perl 5.10 or newer. See "Switch Statements" in perlsyn.
- Variables begin with "$", "@" or "%" in Perl.
- Comments begin with "#", not "/*" or "//". Perl may interpret C/C++ comments as division operators, unterminated regular expressions or the defined-or operator.
- You can't take the address of anything, although a similar operator in Perl is the backslash, which creates a reference.
- `ARGV` must be capitalized. `$ARGV[0]` is C's `argv[1]`, and `argv[0]` ends up in `$0`.
- System calls such as *link()*, *unlink()*, *rename()*, etc. return nonzero for success, not 0. (*system()*, however, returns zero for success.)
- Signal handlers deal with signal names, not numbers. Use `kill -l` to find their names on your system.

JavaScript Traps

Judicious JavaScript programmers should take note of the following:

- In Perl, binary + is always addition. `$string1 + $string2` converts both strings to numbers and then adds them. To concatenate two strings, use the `.` operator.
- The + unary operator doesn't do anything in Perl. It exists to avoid syntactic ambiguities.
- Unlike `for...in`, Perl's `for` (also spelled `foreach`) does not allow the left-hand side to be an arbitrary expression. It must be a variable:

  ```
  for my $variable (keys %hash) {
       ...
  }
  ```

  Furthermore, don't forget the `keys` in there, as `foreach my $kv (%hash) {}` iterates over the keys and values, and is generally not useful ($kv would be a key, then a value, and so on).
- To iterate over the indices of an array, use `foreach my $i (0 .. $#array) {}`. `foreach my $v (@array) {}` iterates over the values.

- Perl requires braces following `if`, `while`, `foreach`, etc.
- In Perl, `else if` is spelled `elsif`.
- `? :` has higher precedence than assignment. In JavaScript, one can write:

  ```
  condition ? do_something() : variable = 3
  ```

  and the variable is only assigned if the condition is false. In Perl, you need parentheses:

  ```
  $condition ? do_something() : ($variable = 3);
  ```

  Or just use `if`.
- Perl requires semicolons to separate statements.
- Variables declared with `my` only affect code *after* the declaration. You cannot write `$x = 1; my $x;` and expect the first assignment to affect the same variable. It will instead assign to an `$x` declared previously in an outer scope, or to a global variable.

  Note also that the variable is not visible until the following *statement*. This means that in `my $x = 1 + $x` the second `$x` refers to one declared previously.
- `my` variables are scoped to the current block, not to the current function. If you write `{my $x;} $x;`, the second `$x` does not refer to the one declared inside the block.
- An object's members cannot be made accessible as variables. The closest Perl equivalent to `with(object) { method() }` is `for`, which can alias `$_` to the object:

  ```
  for ($object) {
      $_->method;
  }
  ```
- The object or class on which a method is called is passed as one of the method's arguments, not as a separate `this` value.

Sed Traps

Seasoned **sed** programmers should take note of the following:

- A Perl program executes only once, not once for each input line. You can do an implicit loop with `-n` or `-p`.
- Backreferences in substitutions use "$" rather than "\".
- The pattern matching metacharacters "(", ")", and "|" do not have backslashes in front.
- The range operator is `...`, rather than comma.

Shell Traps

Sharp shell programmers should take note of the following:

- The backtick operator does variable interpolation without regard to the presence of single quotes in the command.
- The backtick operator does no translation of the return value, unlike **csh**.
- Shells (especially **csh**) do several levels of substitution on each command line. Perl does substitution in only certain constructs such as double quotes, backticks, angle brackets, and search patterns.
- Shells interpret scripts a little bit at a time. Perl compiles the entire program before executing it (except for `BEGIN` blocks, which execute at compile time).
- The arguments are available via `@ARGV`, not `$1`, `$2`, etc.
- The environment is not automatically made available as separate scalar variables.
- The shell's `test` uses "=", "!=", "<" etc for string comparisons and "-eq", "-ne", "-lt" etc for numeric comparisons. This is the reverse of Perl, which uses `eq`, `ne`, `lt` for string comparisons, and `==`, `!=` `<` etc for numeric comparisons.

Perl Traps

Practicing Perl Programmers should take note of the following:

- Remember that many operations behave differently in a list context than they do in a scalar one. See perldata for details.

- Avoid barewords if you can, especially all lowercase ones. You can't tell by just looking at it whether a bareword is a function or a string. By using quotes on strings and parentheses on function calls, you won't ever get them confused.
- You cannot discern from mere inspection which builtins are unary operators (like *chop()* and *chdir()*) and which are list operators (like *print()* and *unlink()*). (Unless prototyped, user-defined subroutines can **only** be list operators, never unary ones.) See perlop and perlsub.
- People have a hard time remembering that some functions default to `$_`, or `@ARGV`, or whatever, but that others which you might expect to do not.
- The <FH> construct is not the name of the filehandle, it is a readline operation on that handle. The data read is assigned to `$_` only if the file read is the sole condition in a while loop:

```
while (<FH>)      { }
while (defined($_ = <FH>)) { }..
<FH>;  # data discarded!
```

- Remember not to use = when you need =~; these two constructs are quite different:

```
$x =  /foo/;
$x =~ /foo/;
```

- The `do {}` construct isn't a real loop that you can use loop control on.
- Use `my()` for local variables whenever you can get away with it (but see perlform for where you can't). Using `local()` actually gives a local value to a global variable, which leaves you open to unforeseen side-effects of dynamic scoping.
- If you localize an exported variable in a module, its exported value will not change. The local name becomes an alias to a new value but the external name is still an alias for the original.

As always, if any of these are ever officially declared as bugs, they'll be fixed and removed.

NAME

perlbook – Books about and related to Perl

DESCRIPTION

There are many books on Perl and Perl-related. A few of these are good, some are OK, but many aren't worth your money. There is a list of these books, some with extensive reviews, at <http://books.perl.org/> . We list some of the books here, and while listing a book implies our endorsement, don't think that not including a book means anything.

Most of these books are available online through Safari Books Online (<http://safaribooksonline.com/>).

The most popular books

The major reference book on Perl, written by the creator of Perl, is *Programming Perl*:

Programming Perl (the "Camel Book"):

```
by Tom Christiansen, brian d foy, Larry Wall with Jon Orwant
ISBN 978-0-596-00492-7 [4th edition February 2012]
ISBN 978-1-4493-9890-3 [ebook]
http://oreilly.com/catalog/9780596004927
```

The Ram is a cookbook with hundreds of examples of using Perl to accomplish specific tasks:

The Perl Cookbook (the "Ram Book"):

```
by Tom Christiansen and Nathan Torkington,
with Foreword by Larry Wall
ISBN 978-0-596-00313-5 [2nd Edition August 2003]
ISBN 978-0-596-15888-0 [ebook]
http://oreilly.com/catalog/9780596003135/
```

If you want to learn the basics of Perl, you might start with the Llama book, which assumes that you already know a little about programming:

Learning Perl (the "Llama Book")

```
by Randal L. Schwartz, Tom Phoenix, and brian d foy
ISBN 978-1-4493-0358-7 [6th edition June 2011]
ISBN 978-1-4493-0458-4 [ebook]
http://www.learning-perl.com/
```

The tutorial started in the Llama continues in the Alpaca, which introduces the intermediate features of references, data structures, object-oriented programming, and modules:

Intermediate Perl (the "Alpaca Book")

```
by Randal L. Schwartz and brian d foy, with Tom Phoenix
        foreword by Damian Conway
ISBN 978-1-4493-9309-0 [2nd edition August 2012]
ISBN 978-1-4493-0459-1 [ebook]
http://www.intermediateperl.com/
```

References

You might want to keep these desktop references close by your keyboard:

Perl 5 Pocket Reference

```
by Johan Vromans
ISBN 978-1-4493-0370-9 [5th edition July 2011]
ISBN 978-1-4493-0813-1 [ebook]
http://oreilly.com/catalog/0636920018476/
```

Perl Debugger Pocket Reference

```
by Richard Foley
ISBN 978-0-596-00503-0 [1st edition January 2004]
ISBN 978-0-596-55625-9 [ebook]
http://oreilly.com/catalog/9780596005030/
```

Regular Expression Pocket Reference

```
by Tony Stubblebine
ISBN 978-0-596-51427-3 [2nd edition July 2007]
ISBN 978-0-596-55782-9 [ebook]
http://oreilly.com/catalog/9780596514273/
```

Tutorials

Beginning Perl

(There are 2 books with this title)

```
by Curtis 'Ovid' Poe
ISBN 978-1-118-01384-7
http://www.wrox.com/WileyCDA/WroxTitle/productCd-1118013840.html

by James Lee
ISBN 1-59059-391-X [3rd edition April 2010 & ebook]
http://www.apress.com/9781430227939
```

Learning Perl (the "Llama Book")

```
by Randal L. Schwartz, Tom Phoenix, and brian d foy
ISBN 978-1-4493-0358-7 [6th edition June 2011]
ISBN 978-1-4493-0458-4 [ebook]
http://www.learning-perl.com/
```

Intermediate Perl (the "Alpaca Book")

```
by Randal L. Schwartz and brian d foy, with Tom Phoenix
        foreword by Damian Conway
ISBN 978-1-4493-9309-0 [2nd edition August 2012]
ISBN 978-1-4493-0459-1 [ebook]
http://www.intermediateperl.com/
```

Mastering Perl

```
   by brian d foy
ISBN 9978-1-4493-9311-3 [2st edition January 2014]
ISBN 978-1-4493-6487-8 [ebook]
http://www.masteringperl.org/
```

Effective Perl Programming

```
by Joseph N. Hall, Joshua A. McAdams, brian d foy
ISBN 0-321-49694-9 [2nd edition 2010]
http://www.effectiveperlprogramming.com/
```

Task-Oriented

Writing Perl Modules for CPAN

```
by Sam Tregar
ISBN 1-59059-018-X [1st edition August 2002 & ebook]
http://www.apress.com/9781590590188
```

The Perl Cookbook

```
by Tom Christiansen and Nathan Torkington,
    with Foreword by Larry Wall
ISBN 978-0-596-00313-5 [2nd Edition August 2003]
ISBN 978-0-596-15888-0 [ebook]
http://oreilly.com/catalog/9780596003135/
```

Automating System Administration with Perl

```
by David N. Blank-Edelman
ISBN 978-0-596-00639-6 [2nd edition May 2009]
ISBN 978-0-596-80251-6 [ebook]
http://oreilly.com/catalog/9780596006396
```

Real World SQL Server Administration with Perl

```
by Linchi Shea
ISBN 1-59059-097-X [1st edition July 2003 & ebook]
http://www.apress.com/9781590590973
```

Special Topics

Regular Expressions Cookbook

by Jan Goyvaerts and Steven Levithan
ISBN 978-1-4493-1943-4 [2nd edition August 2012]
ISBN 978-1-4493-2747-7 [ebook]
http://shop.oreilly.com/product/0636920023630.do

Programming the Perl DBI

by Tim Bunce and Alligator Descartes
ISBN 978-1-56592-699-8 [February 2000]
ISBN 978-1-4493-8670-2 [ebook]
http://oreilly.com/catalog/9781565926998

Perl Best Practices

by Damian Conway
ISBN 978-0-596-00173-5 [1st edition July 2005]
ISBN 978-0-596-15900-9 [ebook]
http://oreilly.com/catalog/9780596001735

Higher-Order Perl

by Mark-Jason Dominus
ISBN 1-55860-701-3 [1st edition March 2005]
free ebook http://hop.perl.plover.com/book/
http://hop.perl.plover.com/

Mastering Regular Expressions

by Jeffrey E. F. Friedl
ISBN 978-0-596-52812-6 [3rd edition August 2006]
ISBN 978-0-596-55899-4 [ebook]
http://oreilly.com/catalog/9780596528126

Network Programming with Perl

by Lincoln Stein
ISBN 0-201-61571-1 [1st edition 2001]
http://www.pearsonhighered.com/educator/product/Network-Programming-with-Perl

Perl Template Toolkit

by Darren Chamberlain, Dave Cross, and Andy Wardley
ISBN 978-0-596-00476-7 [December 2003]
ISBN 978-1-4493-8647-4 [ebook]
http://oreilly.com/catalog/9780596004767

Object Oriented Perl

by Damian Conway
with foreword by Randal L. Schwartz
ISBN 1-884777-79-1 [1st edition August 1999 & ebook]
http://www.manning.com/conway/

Data Munging with Perl

by Dave Cross
ISBN 1-930110-00-6 [1st edition 2001 & ebook]
http://www.manning.com/cross

Mastering Perl/Tk

by Steve Lidie and Nancy Walsh
ISBN 978-1-56592-716-2 [1st edition January 2002]
ISBN 978-0-596-10344-6 [ebook]
http://oreilly.com/catalog/9781565927162

Extending and Embedding Perl

by Tim Jenness and Simon Cozens
ISBN 1-930110-82-0 [1st edition August 2002 & ebook]
http://www.manning.com/jenness

Pro Perl Debugging

```
by Richard Foley with Andy Lester
ISBN 1-59059-454-1 [1st edition July 2005 & ebook]
http://www.apress.com/9781590594544
```

Free (as in beer) books

Some of these books are available as free downloads.

Higher-Order Perl: <http://hop.perl.plover.com/>

Modern Perl: <http://onyxneon.com/books/modern_perl/>

Other interesting, non-Perl books

You might notice several familiar Perl concepts in this collection of ACM columns from Jon Bentley. The similarity to the title of the major Perl book (which came later) is not completely accidental:

Programming Pearls

```
by Jon Bentley
ISBN 978-0-201-65788-3 [2 edition, October 1999]
```

More Programming Pearls

```
by Jon Bentley
ISBN 0-201-11889-0 [January 1988]
```

A note on freshness

Each version of Perl comes with the documentation that was current at the time of release. This poses a problem for content such as book lists. There are probably very nice books published after this list was included in your Perl release, and you can check the latest released version at <http://perldoc.perl.org/perlbook.html> .

Some of the books we've listed appear almost ancient in internet scale, but we've included those books because they still describe the current way of doing things. Not everything in Perl changes every day. Many of the beginner-level books, too, go over basic features and techniques that are still valid today. In general though, we try to limit this list to books published in the past five years.

Get your book listed

If your Perl book isn't listed and you think it should be, let us know. <mailto:perl5-porters@perl.org>