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 **-w** switch it can warn you about this. For example, the third line above produces:

```
print (...) interpreted as function at - line 1.
Useless use of integer addition in void context at - line 1.
```

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

construction version of the comma. That means it was never a list to start with.

In general, functions in Perl that serve as wrappers for system calls ("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 @ARRAYs

`each`, `keys`, `pop`, `push`, `shift`, `splice`, `unshift`, `values`

Functions for list data

`grep`, `join`, `map`, `qw//`, `reverse`, `sort`, `unpack`

Functions for real %HASHes

`delete`, `each`, `exists`, `keys`, `values`

Input and output functions

`binmode`, `close`, `closedir`, `dbmclose`, `dbmopen`, `die`, `eof`, `fileno`, `flock`, `format`, `getc`, `print`, `printf`, `read`, `readdir`, `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`, `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` 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` switches 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;
    $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. See *open*.

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; see *open*. The `:encoding` layer sometimes needs to be called in mid-stream, and it doesn't flush the stream. The `: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 function 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 pragmata. 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.

This keyword is enabled by the `"switch"` feature; see *feature* for more information on `"switch"`. You can also access it by prefixing it with `CORE::`. Alternatively, include a `use v5.10` or later to 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
($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
($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. 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.

```
$cnt = chmod 0755, "foo", "bar";
chmod 0755, @executables;
$mode = "0644"; chmod $mode, "foo";      # !!! sets mode to
                                         # --w----r-T
$mode = "0644"; chmod oct($mode), "foo"; # this is better
$mode = 0644;   chmod $mode, "foo";      # this is best
```

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
    @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($cwd = `pwd`);
chomp($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.

```
$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($user = <STDIN>);
print "Files: ";
chomp($pattern = <STDIN>);

($login,$pass,$uid,$gid) = getpwnam($user)
    or die "$user not in passwd file";

@ary = glob($pattern);  # expand filenames
chown $uid, $gid, @ary;
```

On most systems, you are not allowed to change the ownership of the file unless you're the superuser, although you should be able to change the group to any of your secondary groups. On insecure systems, these restrictions may be relaxed, but this is not a portable assumption. On POSIX systems, you can detect this condition this way:

```
use POSIX qw(sysconf _PC_CHOWN_RESTRICTED);
$can_chown_giveaway = not 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("/")` (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:

```
$pwd = (getpwuid($<))[1];

system "stty -echo";
print "Password: ";
chomp($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 deprecated. It used to report whether memory for that aggregate had ever been allocated. This behavior may disappear in future versions of Perl. 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*.

```
%hash = (foo => 11, bar => 22, baz => 33);
$scalar = delete $hash{foo};         # $scalar is 11
$scalar = delete @hash{qw(foo bar)}; # $scalar is 22
@array  = delete @hash{qw(foo baz)}; # @array  is (undef,33)
```

The following (inefficiently) deletes all the values of %HASH and @ARRAY:

```
foreach $key (keys %HASH) {
    delete $HASH{$key};
}

foreach $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 evals, 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
```

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 `$SIG{__DIE__}` hook. The associated handler is called with the error text and can change the error message, if it sees fit, by calling `die` again. See *"%SIG" in perlvar* for details on setting `%SIG` entries, and *eval BLOCK* for some examples. Although this feature was to be run only right before your program was to exit, this is not currently so: the `$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.

```
do 'stat.pl';
```

is largely like

```
eval `cat stat.pl`;
```

except that it's more concise, runs no external processes, keeps track of the current filename for error messages, searches the `@INC` directories, and updates `%INC` if the file is found. See *"@INC" in perlvar* and *"%INC" in perlvar* for these variables. It also differs in that code evaluated with `do FILENAME` 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.

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
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

each EXPR

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 (($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 (($key,$value) = each %ENV) {
    print "$key=$value\n";
}
```

Starting with Perl 5.14, each can take a scalar EXPR, which must hold a reference to an unblessed hash or array. The argument will be dereferenced automatically. This aspect of each is considered highly experimental. The exact behaviour may change in a future version of Perl.

```
while (($key,$value) = each $hashref) { ... }
```

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 5.014; # so keys/values/each work on scalars (experimental)
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 ungetcs 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

In the first form, often referred to as a "string eval", the return value of EXPR is parsed and executed as if it were a little Perl program. The value of the expression (which is itself determined within scalar context) is first parsed, and if there were no errors, executed as a block within the lexical context of the current Perl program. This means, that in particular, any outer lexical variables are visible to it, and any package variable settings or subroutine and format definitions remain afterwards.

Note that the value is parsed every time the `eval` executes. If EXPR is omitted, evaluates `$_`. This form is typically used to delay parsing and subsequent execution of the text of EXPR until run time.

If the `unicode_eval` feature is enabled (which is the default under a `use 5.16` or higher declaration), EXPR or `$_` is treated as a string of characters, so `use utf8` declarations have no effect, and source filters are forbidden. In the absence of the `unicode_eval` feature, the string will sometimes be treated as characters and sometimes as bytes, depending on the internal encoding, and source filters activated within the `eval` exhibit the erratic, but historical, behaviour of affecting some outer file scope that is still compiling. See also the *evalbytes* keyword, which always treats its input as a byte stream and works properly with source filters, and the *feature* pragma.

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.

In the second form, the code within the BLOCK is parsed only once--at the same time the code surrounding the `eval` itself was parsed--and executed within the context of the current

Perl program. This form is typically used to trap exceptions more efficiently than the first (see below), while also providing the benefit of checking the code within BLOCK at compile time.

The final semicolon, if any, may be omitted from the value of EXPR or within the BLOCK.

In both forms, the value returned is the value of the last expression evaluated inside the mini-program; a return statement may be also used, just as with subroutines. The expression providing the return value is evaluated in void, scalar, or list context, depending on the context of the `eval` itself. See *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.

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*.

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 $@
```

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.

With an `eval`, you should be especially careful to remember what's being looked at when:

```
eval $x;        # CASE 1
eval "$x";      # CASE 2

eval '$x';      # CASE 3
eval { $x };    # CASE 4

eval "\$$x++";  # CASE 5
$$x++;          # CASE 6
```

Cases 1 and 2 above behave identically: they run the code contained in the variable $x. (Although case 2 has misleading double quotes making the reader wonder what else might be happening (nothing is).) Cases 3 and 4 likewise behave in the same way: they run the code `'$x'`, which does nothing but return the value of $x. (Case 4 is preferred for purely visual reasons, but it also has the advantage of compiling at compile-time instead of at run-time.) Case 5 is a place where normally you *would* like to use double quotes, except that in this particular situation, you can just use symbolic references instead, as in case 6.

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
}
```

`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.

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.

evalbytes EXPR

evalbytes

This function is like *eval* with a string argument, except it always parses its argument, or `$_` if EXPR is omitted, as a string of bytes. A string containing characters whose ordinal value exceeds 255 results in an error. Source filters activated within the evaluated code apply to the code itself.

This function is only available under the `evalbytes` feature, a `use v5.16` (or higher) declaration, or with a `CORE::` prefix. See *feature* for more information.

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 `-w` is set--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:

```
$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.

```
@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:

```
$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 pragma, 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.

This keyword is available only when the `"fc"` feature is enabled, or when prefixed with `CORE::`; See *feature*. Alternately, include a `use v5.16` or later to 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;
fcntl($filehandle, F_GETFL, $packed_return_buffer)
    or die "can't fcntl F_GETFL: $!";
```

You don't have to check for `defined` on the return from `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 **-w** 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);

$flags = fcntl(REMOTE, F_GETFL, 0)
            or die "Can't get flags for the socket: $!\n";

$flags = 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(), 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() 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(). 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 $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";
    }

    $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 from your nearest *CPAN* site.

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`.

```
$login = getlogin || getpwuid($<) || "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;
$hersockaddr    = getpeername(SOCK);
($port, $iaddr) = sockaddr_in($hersockaddr);
$herhostname    = gethostbyaddr($iaddr, AF_INET);
$herstraddr     = inet_ntoa($iaddr);
```

getpgrp PID

Returns the current process group for the specified PID. Use a PID of `0` to get the current process group for the current process. Will raise an exception if used on a machine that doesn't implement getpgrp(2). If PID is omitted, returns 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
( $name,    $passwd,    $gid,        $members  ) = getgr*
( $name,    $aliases,   $addrtype,   $net      ) = getnet*
( $name,    $aliases,   $port,       $proto    ) = getserv*
( $name,    $aliases,   $proto                 ) = getproto*
( $name,    $aliases,   $addrtype,   $length,  @addrs ) = gethost*
( $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:

```
$uid   = getpwnam($name);
$name  = getpwuid($num);
$name  = getpwent();
$gid   = getgrnam($name);
$name  = getgrgid($num);
$name  = getgrent();
#etc.
```

In *getpw*()* the fields $quota, $comment, and $expire are special 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:

```
($a,$b,$c,$d) = unpack('W4',$addr[0]);
```

The Socket library makes this slightly easier:

```
use Socket;
$iaddr = inet_aton("127.1"); # or whatever address
$name  = gethostbyaddr($iaddr, AF_INET);

# or going the other way
$straddr = inet_ntoa($iaddr);
```

In the opposite way, to resolve a hostname to the IP address you can write this:

```
use Socket;
$packed_ip = gethostbyname("www.perl.org");
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;
$is_his = (stat($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;
$mysockaddr = getsockname(SOCK);
($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 either of:

```
@spacies = <"*e f*">;
@spacies = glob '"*e f*"';
@spacies = glob q("*e f*");
```

If you had to get a variable through, you could do this:

```
@spacies = glob "'*${var}e f*'";
@spacies = glob qq("*${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:

```
@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.

```
    @foo = grep(!/^#/, @bar);    # weed out comments
```

or equivalently,

```
    @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.

If $_ is lexical in the scope where the `grep` appears (because it has been declared with the deprecated `my $_` construct) then, in addition to being locally aliased to the list elements, $_ keeps being lexical inside the block; i.e., it can't be seen from the outside, avoiding any potential side-effects.

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 value. (To convert strings that might start with either `0`, `0x`, or `0b`, see *oct*.) If EXPR is omitted, uses $_.

```
    print hex '0xAf'; # prints '175'
    print hex 'aF';   # same
```

Hex strings may only represent integers. Strings that would cause integer overflow trigger a warning. Leading whitespace is not stripped, unlike 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:

```
$retval = ioctl(...) || -1;
printf "System returned %d\n", $retval;
```

The special string `"0 but true"` is exempt from **-w** complaints about 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:

```
$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

keys EXPR

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:

```
@keys = keys %ENV;
@values = values %ENV;
while (@keys) {
    print pop(@keys), '=', pop(@values), "\n";
}
```

or how about sorted by key:

```
foreach $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 $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, `keys` can take a scalar EXPR, which must contain a reference to an unblessed hash or array. The argument will be dereferenced automatically. This aspect of `keys` is considered highly experimental. The exact behaviour may change in a future version of Perl.

```
for (keys $hashref) { ... }
```

```
for (keys $obj->get_arrayref) { ... }
```

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 5.014; # so keys/values/each work on scalars (experimental)
```

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).

```
$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() 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 pragmata, 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_utf8(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
($sec,$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(3)).

In scalar context, localtime() returns the ctime(3) value:

```
$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(3) and mktime(3) 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);
$now_string = strftime "%a %b %e %H:%M:%S %Y", localtime;
# or for GMT formatted appropriately for your locale:
$now_string = strftime "%a %b %e %H:%M:%S %Y", gmtime;
```

Note that the %a and %b, the short forms of the day of the week and the month of the year, may not necessarily be three characters wide.

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.

```
@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.

```
%hash = map { get_a_key_for($_) => $_ } @array;
```

is just a funny way to write

```
%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 an array composed of those items of the original list for which the BLOCK or EXPR evaluates to true.

If `$_` is lexical in the scope where the `map` appears (because it has been declared with the deprecated `my $_` construct), then, in addition to being locally aliased to the list elements, `$_` keeps being lexical inside the block; that is, it can't be seen from the outside, avoiding any potential side-effects.

`{` 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:

```
%hash = map {  "\L$_" => 1  } @array # perl guesses EXPR. wrong
%hash = map { +"\L$_" => 1  } @array # perl guesses BLOCK. right
%hash = map {; "\L$_" => 1  } @array # this also works
%hash = map { ("\L$_" => 1) } @array # as does this
%hash = map {  lc($_) => 1  } @array # and this.
%hash = map +( lc($_) => 1 ), @array # this is EXPR and works!

%hash = map  ( lc($_), 1 ),   @array # evaluates to (1, @array)
```

or to force an anon hash constructor use `+{`:

```
@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 perlfunc(1) entry on `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::SysV::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 the `IPC::SysV` and `IPC::SysV::Msg` documentation.

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, and *fields*, *attributes*, and *Attribute::Handlers*.

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():

```
$dec_perms = (stat("filename"))[2] & 07777;
$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 "cannot open < input.txt: $!";
```

and for writing:

```
    open(my $fh, ">", "output.txt")
```

```
or die "cannot 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 always 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 specifying 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.

If you're running Perl on a system that distinguishes between text files and binary files, then you should check out *binmode* for tips for dealing with this. The key distinction between

systems that need `binmode` and those that don't is their text file formats. Systems like Unix, Mac OS, and Plan 9, that end lines with a single character and encode that character in C as `"\n"` do not need `binmode`. The rest need it.

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 "cannot 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 an anonymous temporary file. Also using +< works for symmetry, but you really should consider writing something to the temporary file first. 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($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: $!";
```

General examples:

```
open(LOG, ">>/usr/spool/news/twitlog");  # (log is reserved)
# 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(ARTICLE, "-|", "caesar <$article")  # decrypt article
    or die "Can't start caesar: $!";

open(ARTICLE, "caesar <$article |")      # ditto
    or die "Can't start caesar: $!";

open(EXTRACT, "|sort >Tmp$$")            # $$ is our process id
    or die "Can't start sort: $!";

# in-memory files
open(MEMORY, ">", \$var)
    or die "Can't open memory file: $!";
print MEMORY "foo!\n";              # output will appear in $var

# process argument list of files along with any includes

foreach $file (@ARGV) {
    process($file, "fh00");
}

sub process {
    my($filename, $input) = @_;
    $input++;    # this is a string increment
    unless (open($input, "<", $filename)) {
        print STDERR "Can't open $filename: $!\n";
        return;
    }

    local $_;
    while (<$input>) {    # note use of indirection
        if (/^#include "(.*)"/) {
            process($1, $input);
            next;
        }
        #...          # whatever
    }
}
```

See *perliol* for detailed info on PerlIO.

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 `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` 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 $fd
    open(FILEHANDLE, "<&=$fd")
```

or

```
    open(FILEHANDLE, "<&=", $fd)
```

or

```
    # open for append, using the fileno of OLDFH
    open(FH, ">>&=", OLDFH)
```

or

```
    open(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(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(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() to implement the = functionality. On many Unix systems, fdopen() 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` and looking for the `useperlio=` line. If `useperlio` is `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

```
    $child_pid = open(FROM_KID, "-|")  // die "can't fork: $!";
```

or

```
    $child_pid = open(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(FOO, "|tr '[a-z]' '[A-Z]'");
    open(FOO, "|-", "tr '[a-z]' '[A-Z]'");
    open(FOO, "|-") || exec 'tr', '[a-z]', '[A-Z]';
    open(FOO, "|-", "tr", '[a-z]', '[A-Z]');

    open(FOO, "cat -n '$file'|");
    open(FOO, "-|", "cat -n '$file'");
    open(FOO, "-|") || exec "cat", "-n", $file;
    open(FOO, "-|", "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(FOO, "|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(FH, $filename) or die "Can't open $filename: $!";
```

Use the three-argument form to open a file with arbitrary weird characters in it,

```
    open(FOO, "<", $file)
```

```
 || die "can't open < $file: $!";
```

otherwise it's necessary to protect any leading and trailing whitespace:

```
    $file =~ s#^(\s)#./$1#;
    open(FOO, "< $file\0")
 || die "open failed: $!";
```

(this may not work on some bizarre filesystems). One should conscientiously choose between the *magic* and *three-argument* form of open():

```
    open(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(IN, "<", $ARGV[0])
 || die "can't open < $ARGV[0]: $!";
```

will have exactly the opposite restrictions.

If you want a "real" C `open` (see *open(2)* on your system), then you should use the `sysopen` function, which involves no such magic (but may use subtly different filemodes than Perl open(), which is mapped to C fopen()). This is another way to protect your filenames from interpretation. For example:

```
    use IO::Handle;
    sysopen(HANDLE, $path, O_RDWR|O_CREAT|O_EXCL)
        or die "sysopen $path: $!";
    $oldfh = select(HANDLE); $| = 1; select($oldfh);
    print HANDLE "stuff $$\n";
    seek(HANDLE, 0, 0);
    print "File contains: ", <HANDLE>;
```

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, and *fields*, *attributes*, and *Attribute::Handlers* .

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.

Larry recalls that the hex and bit string formats (H, h, B, b) were added to pack for processing data from NASA's Magellan probe. Magellan was in an elliptical orbit, using the antenna for the radar mapping when close to Venus and for communicating data back to Earth for the rest of the orbit. There were two transmission units, but one of these failed, and then the other developed a fault whereby it would randomly flip the sense of all the bits. It was easy to automatically detect complete records with the correct sense, and complete records with all the bits flipped. However, this didn't recover the records where the sense flipped midway. A colleague of Larry's was able to pretty much eyeball where the records flipped, so they wrote an editor named kybble (a pun on the dog food Kibbles 'n Bits) to enable him to manually correct the records and recover the data. For this purpose pack gained the hex and bit string format specifiers.

git shows that they were added to perl 3.0 in patch #44 (Jan 1991, commit 27e2fb84680b9cc1), but the patch description makes no mention of their addition, let alone the story behind them.

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 EXPR

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, `pop` can take a scalar EXPR, which must hold a reference to an unblessed array. The argument will be dereferenced automatically. This aspect of `pop` is considered highly experimental. The exact behaviour may change in a future version of Perl.

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 5.014; # so push/pop/etc work on scalars (experimental)
```

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). 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 real 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 for LC_NUMERIC Look for this throught pod 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 real 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

push EXPR,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 $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, `push` can take a scalar EXPR, which must hold a reference to an unblessed array. The argument will be dereferenced automatically. This aspect of `push` is considered highly experimental. The exact behaviour may change in a future version of Perl.

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 5.014; # so push/pop/etc work on scalars (experimental)
```

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, *open*), the I/O will operate on UTF8-encoded Unicode characters, not bytes. Similarly for the `:encoding` pragma: 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: $!";
@dots = grep { /^\./ && -f "$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;
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`). 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*.

```
$line = <STDIN>;
$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( $_ = <$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( $_ = <$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`). 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 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(utf8)` I/O layer (see the `open` pragma, *open*), the I/O will operate on UTF8-encoded Unicode characters, not bytes. Similarly for the `:encoding` pragma: 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|{.*| |) {
        $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 (aka $PERL_VERSION). 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, the require assumes a "*.pm*" extension and replaces "*::*" with "*/*" in the filename for you, to make it easy to load standard modules. This form of loading of modules does not risk altering your namespace.

In other words, if you try this:

```
        require Foo::Bar;    # a splendid bareword
```

The require function will actually look for the "*Foo/Bar.pm*" file in the directories specified in the `@INC` array.

But if you try this:

```
        $class = 'Foo::Bar';
        require $class;      # $class is not a bareword
    #or
        require "Foo::Bar";  # not a bareword because of the ""
```

The require function will look for the "*Foo::Bar*" file in the @INC array and will complain about not finding "*Foo::Bar*" there. In this case you can do:

```
        eval "require $class";
```

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, $y, ...
        my @parameters = @$arrayref[1..$#$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 ?? searches so that they work again. The expression is interpreted as a list of single characters (hyphens allowed for ranges). All variables and arrays beginning with one of those letters are reset to their pristine state. If the expression is omitted, one-match searches (`?pattern?`) are reset to match again. 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 ?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`, or `do FILE` with the value given in EXPR. Evaluation of EXPR may be in list, scalar, or void context, depending on how the return value will be used,

and the context may vary from one execution to the next (see *wantarray*). If no EXPR is given, returns an empty list in list context, 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.

```
%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 real filehandle like `FH`, not an indirect one like `$fh`.

This keyword is available only when the `"say"` feature is enabled, or when prefixed with `CORE::`; see *feature*. Alternately, include a `use v5.10` or later to the current scope.

scalar EXPR

Forces EXPR to be interpreted in scalar context and returns the value of EXPR.

```
@counts = ( scalar @a, scalar @b, scalar @c );
```

There is no equivalent operator to force an expression to be interpolated in list context because 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.

seek FILEHANDLE,POSITION,WHENCE

Sets FILEHANDLE's position, just like the `fseek` call of `stdio`. FILEHANDLE may be an expression whose value gives the name of the filehandle. The values for WHENCE are `0` to set the new position *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 *in bytes*: even if the filehandle has been set to operate on characters (for example by using the `:encoding(utf8)` open layer), tell() will return byte offsets, not character offsets (because implementing that would render seek() and tell() rather slow).

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(TEST,0,1);
```

This is also useful for applications emulating `tail -f`. Once you hit EOF on your read and then sleep for a while, you (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 `<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(FILE); $_ = <FILE>;
         $curpos = tell(FILE)) {
        # search for some stuff and put it into files
    }
    sleep($for_a_while);
    seek(FILE, $curpos, 0);
}
```

seekdir DIRHANDLE,POS

Sets the current position for the `readdir` routine on DIRHANDLE. POS must be a value returned by `telldir`. `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:

```
$oldfh = select(STDERR); $| = 1; select($oldfh);
```

Some programmers may prefer to think of filehandles as objects with methods, preferring to write the last example as:

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

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:

```
$rin = $win = $ein = '';
vec($rin, fileno(STDIN),  1) = 1;
vec($win, fileno(STDOUT), 1) = 1;
$ein = $rin | $win;
```

If you want to select on many filehandles, you 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;
    }
    $rin = fhbits(*STDIN, *TTY, *MYSOCK);
```

The usual idiom is:

```
    ($nfound,$timeleft) =
      select($rout=$rin, $wout=$win, $eout=$ein, $timeout);
```

or to block until something becomes ready just do this

```
    $nfound = select($rout=$rin, $wout=$win, $eout=$ein, undef);
```

Most systems do not bother to return anything useful in $timeleft, so calling select() in scalar context just returns $nfound.

Any of the bit masks can also be undef. The timeout, if specified, is in seconds, which may be fractional. Note: not all implementations are capable of returning the $timeleft. If not, they always return $timeleft equal to the supplied $timeout.

You can effect a sleep of 250 milliseconds this way:

```
    select(undef, undef, undef, 0.25);
```

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 <FH>) 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*, `IPC::SysV`, `IPC::Semaphore` documentation.

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*, `IPC::SysV`, `IPC::SysV::Semaphore` documentation.

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:

```
$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*, `IPC::SysV`, and `IPC::SysV::Semaphore` documentation.

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(utf8)` I/O layer (see *open*, or the `open` pragma, *open*), the I/O will operate on UTF-8 encoded Unicode characters, not bytes. Similarly for the `:encoding` pragma: 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 EXPR

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, `shift` can take a scalar EXPR, which must hold a reference to an unblessed array. The argument will be dereferenced automatically. This aspect of `shift` is considered highly experimental. The exact behaviour may change in a future version of Perl.

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 5.014; # so push/pop/etc work on scalars (experimental)
```

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 `IPC::SysV` documentation.

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 `IPC::SysV` documentation.

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*, `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{ALARM} = 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 defined `pipe` in terms of `socketpair`, in which a call to `pipe(Rdr, Wtr)` is essentially:

```
use Socket;
socketpair(Rdr, Wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC);
shutdown(Rdr, 1);        # no more writing for reader
shutdown(Wtr, 0);        # no more reading for writer
```

See *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). Note that in the latter case, it is usually highly counter-productive to declare $a and $b as lexicals.

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, so *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. See *the sort pragma*.

Examples:

```
# sort lexically
@articles = sort @files;

# same thing, but with explicit sort routine
@articles = sort {$a cmp $b} @files;

# now case-insensitively
@articles = sort {fc($a) cmp fc($b)} @files;
```

```
# same thing in reversed order
@articles = sort {$b cmp $a} @files;

# sort numerically ascending
@articles = sort {$a <=> $b} @files;

# sort numerically descending
@articles = sort {$b <=> $a} @files;

# this sorts the %age hash by value instead of key
# using an in-line function
@eldest = sort { $age{$b} <=> $age{$a} } keys %age;

# sort using explicit subroutine name
sub byage {
    $age{$a} <=> $age{$b};  # presuming numeric
}
@sortedclass = sort byage @class;

sub backwards { $b cmp $a }
@harry  = qw(dog cat x Cain Abel);
@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
```

```
    @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;
    @new = sort other::backwards @old;

    # guarantee stability, regardless of algorithm
    use sort 'stable';
    @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 _
    @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:

```
    @contact = sort { $a cmp $b } find_records @key;
    @contact = sort +find_records(@key);
    @contact = sort &find_records(@key);
    @contact = sort(find_records(@key));
```

If instead you want to sort the array @key with the comparison routine `find_records()` then you can use:

```
    @contact = sort { find_records() } @key;
    @contact = sort find_records(@key);
    @contact = sort(find_records @key);
    @contact = sort(find_records (@key));
```

If you're using strict, you *must not* declare $a and $b as lexicals. They are package globals. That means that if you're in the `main` package and type

```
    @articles = sort {$b <=> $a} @files;
```

then $a and $b are `$main::a` and `$main::b` (or `$::a` and `$::b`), but if you're in the `FooPack` package, it's the same as typing

```
    @articles = sort {$FooPack::b <=> $FooPack::a} @files;
```

The comparison function is required to behave. If it returns inconsistent results (sometimes saying `$x[1]` is less than `$x[2]` and sometimes saying the opposite, for example) the results are not well-defined.

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.

```
    @result = sort { $a <=> $b } grep { $_ == $_ } @input;
```

splice ARRAY,OFFSET,LENGTH,LIST

splice ARRAY,OFFSET,LENGTH

splice ARRAY,OFFSET

splice ARRAY

splice EXPR,OFFSET,LENGTH,LIST

splice EXPR,OFFSET,LENGTH

splice EXPR,OFFSET

splice EXPR

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, `splice` can take scalar EXPR, which must hold a reference to an unblessed array. The argument will be dereferenced automatically. This aspect of `splice` is considered highly experimental. The exact behaviour may change in a future version of Perl.

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 5.014; # so push/pop/etc work on scalars (experimental)
```

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.

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.

As another special case, `split` emulates the default behavior of the command line tool **awk** when the PATTERN is either omitted or a *literal 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 as 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:

```
($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
    $result = sprintf("%08d", $number);

    # Round number to 3 digits after decimal point
    $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>',  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, a plus sign is used to prefix a positive number.

```
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 digits to show, including those prior to the decimal point and those after it; 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>"
```

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 `.*`:

```
printf '<%.6x>', 1;       # prints "<000001>"
printf '<%.*x>', 6, 1;    # prints "<000001>"
```

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>"
```

You cannot currently get the precision from a specified number, but it is intended that this will be possible in the future, for example using `.*2$`:

```
printf '<%.*2$x>', 1, 6;   # INVALID, but in future will print
                           # "<000001>"
```

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"
```

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:

```
($dev,$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.

```
$mode = (stat($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;
$sb = stat($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';

$mode = (stat($filename))[2];

$user_rwx      = ($mode & S_IRWXU) >> 6;
$group_read    = ($mode & S_IRGRP) >> 3;
$other_execute =  $mode & S_IXOTH;

printf "Permissions are %04o\n", S_IMODE($mode), "\n";

$is_setuid     =  $mode & S_ISUID;
$is_directory  =  S_ISDIR($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` variables are enabled only when the `use feature "state"` pragma is in effect, unless the keyword is written as `CORE::state`. See also *feature*. Alternately, include a `use v5.10` or later to the current scope.

study SCALAR

study

May take extra time to study SCALAR ($_ if unspecified) in anticipation of doing many pattern matches on the string before it is next modified. This may or may not save time, depending on the nature and number of patterns you are searching and the distribution of character frequencies in the string to be searched; you probably want to compare run times with and without it to see which is faster. Those loops that scan for many short constant strings (including the constant parts of more complex patterns) will benefit most.

Note that since Perl version 5.16 this function has been a no-op, but this might change in a future release.

(The way `study` works is this: a linked list of every character in the string to be searched is made, so we know, for example, where all the `'k'` characters are. From each search string, the rarest character is selected, based on some static frequency tables constructed from some C programs and English text. Only those places that contain this "rarest" character are examined.)

For example, here is a loop that inserts index producing entries before any line containing a certain pattern:

```
while (<>) {
    study;
    print ".IX foo\n"    if /\bfoo\b/;
    print ".IX bar\n"    if /\bbar\b/;
    print ".IX blurfl\n" if /\bblurfl\b/;
    # ...
    print;
}
```

In searching for `/\bfoo\b/`, only locations in `$_` that contain `f` will be looked at, because `f` is rarer than `o`. In general, this is a big win except in pathological cases. The only question is whether it saves you more time than it took to build the linked list in the first place.

Note that if you have to look for strings that you don't know till runtime, you can build an entire loop as a string and `eval` that to avoid recompiling all your patterns all the time. Together with

undefining $/ to input entire files as one record, this can be quite fast, often faster than specialized programs like fgrep(1). The following scans a list of files (@files) for a list of words (@words), and prints out the names of those files that contain a match:

```
$search = 'while (<>) { study;';
foreach $word (@words) {
    $search .= "++\$seen{\$ARGV} if /\\b$word\\b/;\n";
}
$search .= "}";
@ARGV = @files;
undef $/;
eval $search;        # this screams
$/ = "\n";        # put back to normal input delimiter
foreach $file (sort keys(%seen)) {
    print $file, "\n";
}
```

sub 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:

```
$x = '1234';
for (substr($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:

```
$x = '1234';
for (substr($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:

```
$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
$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.

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 the perlfunc(1) entry on `umask` for more on this.

Note that `sysopen` depends on the fdopen() C library function. On many Unix systems, fdopen() 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.

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 the 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 buffers 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, *open*.

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 *in bytes*: even if the filehandle has been set to operate on characters (for example by using the `:encoding(utf8)` I/O layer), tell() will return byte offsets, not character offsets (because implementing that would render sysseek() unacceptably slow).

sysseek() bypasses normal buffered IO, so mixing it with reads other than `sysread` (for example <> 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 LIST`, 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.

```
@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, *open*.

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 *in bytes*: even if the filehandle has been set to operate on characters (for example by using the `:encoding(utf8)` open layer), tell() will return byte offsets, not character offsets (because that would render seek() and tell() rather slow).

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()` 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(%HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

A class implementing a hash should have the following methods:

```
TIEHASH classname, LIST
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 VARIABLE*.

tied VARIABLE

Returns a reference to the object underlying VARIABLE (the same value that was originally returned by the `tie` call that bound the variable to a package.) Returns the undefined value if VARIABLE isn't tied to a package.

time

Returns the number of non-leap seconds since whatever time the system considers to be the epoch, 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.

```
($user,$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 pragma, 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 pragma, 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 $they_blew_it;
select undef, undef, undef, 0.25;
($a, $b, undef, $c) = &foo;       # Ignore third value returned
```

Note that this is a unary operator, not a list operator.

unlink LIST

unlink

Deletes a list of files. 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. 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:

```
$checksum = do {
    local $/;  # slurp!
    unpack("%32W*",<>) % 65535;
};
```

The following efficiently counts the number of set bits in a bit vector:

```
$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

unshift EXPR,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, unshift can take a scalar EXPR, which must hold a reference to an unblessed array. The argument will be dereferenced automatically. This aspect of unshift is considered highly experimental. The exact behaviour may change in a future version of Perl.

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 5.014; # so push/pop/etc work on scalars (experimental)
```

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. 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. 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 `unimport Module 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
$atime = $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 $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

values EXPR

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, `values` can take a scalar EXPR, which must hold a reference to an unblessed hash or array. The argument will be dereferenced automatically. This aspect of `values` is considered highly experimental. The exact behaviour may change in a future version of Perl.

```
for (values $hashref) { ... }
for (values $obj->get_arrayref) { ... }
```

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 5.014; # so keys/values/each work on scalars (experimental)
```

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), this is ignored by `vec`, and it operates on the internal byte string, not the conceptual character string, even if you only have characters with values less than 256.

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:

```
$bits = unpack("b*", $vector);
@bits = split(//, unpack("b*", $vector));
```

If you know the exact length in bits, it can be used in place of the `*`.

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($_, 3, 2) = 1   ==         64 00000100000000000000000000000000
vec($_, 4, 2) = 1   ==        256 00000001000000000000000000000000
vec($_, 5, 2) = 1   ==       1024 00000000010000000000000000000000
vec($_, 6, 2) = 1   ==       4096 00000000000100000000000000000000
vec($_, 7, 2) = 1   ==      16384 00000000000001000000000000000000
vec($_, 8, 2) = 1   ==      65536 00000000000000010000000000000000
vec($_, 9, 2) = 1   ==     262144 00000000000000000100000000000000
vec($_,10, 2) = 1   ==    1048576 00000000000000000001000000000000
```

```
vec($_,11, 2) = 1   ==    4194304 00000000000000000000001000000000
vec($_,12, 2) = 1   ==   16777216 00000000000000000000010000000000
vec($_,13, 2) = 1   ==   67108864 00000000000000000000100000000000
vec($_,14, 2) = 1   ==  268435456 00000000000000000001000000000000
vec($_,15, 2) = 1   == 1073741824 00000000000000000010000000000000
vec($_, 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 00000000100000000000000000000000
vec($_, 5, 2) = 2   ==       2048 00000000001000000000000000000000
vec($_, 6, 2) = 2   ==       8192 00000000000010000000000000000000
vec($_, 7, 2) = 2   ==      32768 00000000000000100000000000000000
vec($_, 8, 2) = 2   ==     131072 00000000000000001000000000000000
vec($_, 9, 2) = 2   ==     524288 00000000000000000100000000000000
vec($_,10, 2) = 2   ==    2097152 00000000000000000010000000000000
vec($_,11, 2) = 2   ==    8388608 00000000000000000001000000000000
vec($_,12, 2) = 2   ==   33554432 00000000000000000000100000000000
vec($_,13, 2) = 2   ==  134217728 00000000000000000000010000000000
vec($_,14, 2) = 2   ==  536870912 00000000000000000000001000000000
vec($_,15, 2) = 2   == 2147483648 00000000000000000000000100000000
vec($_, 0, 4) = 1   ==          1 10000000000000000000000000000000
vec($_, 1, 4) = 1   ==         16 00001000000000000000000000000000
vec($_, 2, 4) = 1   ==        256 00000000100000000000000000000000
vec($_, 3, 4) = 1   ==       4096 00000000000010000000000000000000
vec($_, 4, 4) = 1   ==      65536 00000000000000001000000000000000
vec($_, 5, 4) = 1   ==    1048576 00000000000000000000100000000000
vec($_, 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($_, 3, 4) = 2   ==       8192 00000000000010000000000000000000
vec($_, 4, 4) = 2   ==     131072 00000000000000001000000000000000
vec($_, 5, 4) = 2   ==    2097152 00000000000000000010000000000000
vec($_, 6, 4) = 2   ==   33554432 00000000000000000000001000000000
vec($_, 7, 4) = 2   ==  536870912 00000000000000000000000000000100
vec($_, 0, 4) = 4   ==          4 00100000000000000000000000000000
vec($_, 1, 4) = 4   ==         64 00000010000000000000000000000000
vec($_, 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 00000000001000000000000000000000
vec($_, 3, 4) = 8   ==      32768 00000000000000001000000000000000
vec($_, 4, 4) = 8   ==     524288 00000000000000000100000000000000
vec($_, 5, 4) = 8   ==    8388608 00000000000000000001000000000000
vec($_, 6, 4) = 8   ==  134217728 00000000000000000000010000000000
vec($_, 7, 4) = 8   == 2147483648 00000000000000000000000100000000
vec($_, 0, 8) = 1   ==          1 10000000000000000000000000000000
vec($_, 1, 8) = 1   ==        256 00000000100000000000000000000000
vec($_, 2, 8) = 1   ==      65536 00000000000000001000000000000000
```

```
vec($_, 3, 8) = 1   ==   16777216 00000000000000000000000010000000
vec($_, 0, 8) = 2   ==          2 01000000000000000000000000000000
vec($_, 1, 8) = 2   ==        512 00000000100000000000000000000000
vec($_, 2, 8) = 2   ==     131072 00000000000000001000000000000000
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($_, 3, 8) = 4   ==   67108864 00000000000000000000000100000000
vec($_, 0, 8) = 8   ==          8 00010000000000000000000000000000
vec($_, 1, 8) = 8   ==       2048 00000000000100000000000000000000
vec($_, 2, 8) = 8   ==     524288 00000000000000000001000000000000
vec($_, 3, 8) = 8   ==  134217728 00000000000000000000000010000000
vec($_, 0, 8) = 16  ==         16 00001000000000000000000000000000
vec($_, 1, 8) = 16  ==       4096 00000000000010000000000000000000
vec($_, 2, 8) = 16  ==    1048576 00000000000000000100000000000000
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 00000010000000000000000000000000
vec($_, 1, 8) = 64  ==      16384 00000000000000100000000000000000
vec($_, 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. On some systems, a value of 0 indicates that there are processes still running. The status is returned in `$?` and `${^CHILD_ERROR_NATIVE}`. If you say

```
use POSIX ":sys_wait_h";
#...
do {
    $kid = waitpid(-1, WNOHANG);
} while $kid > 0;
```

then you can do a non-blocking wait for all pending zombie processes. 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*.