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 a scalar context to its argument, while a list operator may provide either scalar or list contexts for its arguments. If it does both, the scalar arguments will be first, and the list argument will follow. (Note that there can ever be only one 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 the elements of the list) are shown with LIST as an argument. Such a list may consist of any combination of scalar arguments or list values; the list values will be included in the list as if each individual element were interpolated at that point in the list, forming a longer single-dimensional list value. Commas should separate 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 the parentheses, the simple (but occasionally surprising) rule is this: It *looks* like a function, therefore it *is* a function, and precedence doesn't matter. Otherwise it's a list operator or unary operator, and precedence does matter. And whitespace between the function and left parenthesis doesn't count--so you need to be careful sometimes:

```
print 1+2+4; # Prints 7.
print(1+2) + 4; # Prints 3.
print (1+2)+4; # Also prints 3!
print +(1+2)+4; # Prints 7.
print ((1+2)+4); # Prints 7.
```

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

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

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 a scalar context by returning the undefined value, and in a list context by returning the null list.

Remember the following important rule: There is **no rule** that relates the behavior of an expression in list context to its behavior in scalar context, or vice versa. It might do two totally different things. Each operator and function decides which sort of value it would be most appropriate to return in 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 of the same name (like chown(2), fork(2), closedir(2), etc.) all return true when they succeed and `undef` otherwise, as is usually mentioned in the descriptions below. This is different from the C interfaces, which return `-1` on failure. Exceptions to this rule are `wait`, `waitpid`, and `syscall`. System calls also set the special `$!` variable on failure. Other functions do not, except accidentally.

Perl Functions by Category

Here are Perl's functions (including things that look like functions, like some keywords and named operators) arranged by category. Some functions appear in more than one place.

Functions for SCALARs or strings

> `chomp`, `chop`, `chr`, `crypt`, `hex`, `index`, `lc`, `lcfirst`, `length`, `oct`, `ord`, `pack`, `q/STRING/`, `qq/STRING/`, `reverse`, `rindex`, `sprintf`, `substr`, `tr///`, `uc`, `ucfirst`, `y///`

Regular expressions and pattern matching

> `m//`, `pos`, `quotemeta`, `s///`, `split`, `study`, `qr//`

Numeric functions

> `abs`, `atan2`, `cos`, `exp`, `hex`, `int`, `log`, `oct`, `rand`, `sin`, `sqrt`, `srand`

Functions for real @ARRAYs

> `pop`, `push`, `shift`, `splice`, `unshift`

Functions for list data

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

Functions for real %HASHes

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

Input and output functions

> `binmode`, `close`, `closedir`, `dbmclose`, `dbmopen`, `die`, `eof`, `fileno`, `flock`, `format`, `getc`, `print`, `printf`, `read`, `readdir`, `rewinddir`, `seek`, `seekdir`, `select`, `syscall`, `sysread`, `sysseek`, `syswrite`, `tell`, `telldir`, `truncate`, `warn`, `write`

Functions for fixed length data or records

> `pack`, `read`, `syscall`, `sysread`, `syswrite`, `unpack`, `vec`

Functions for filehandles, files, or directories

> `-`*`X`*, `chdir`, `chmod`, `chown`, `chroot`, `fcntl`, `glob`, `ioctl`, `link`, `lstat`, `mkdir`, `open`, `opendir`, `readlink`, `rename`, `rmdir`, `stat`, `symlink`, `sysopen`, `umask`, `unlink`, `utime`

Keywords related to the control flow of your Perl program

> `caller`, `continue`, `die`, `do`, `dump`, `eval`, `exit`, `goto`, `last`, `next`, `redo`, `return`, `sub`, `wantarray`

Keywords related to scoping

> `caller`, `import`, `local`, `my`, `our`, `package`, `use`

Miscellaneous functions

> `defined`, `dump`, `eval`, `formline`, `local`, `my`, `our`, `reset`, `scalar`, `undef`, `wantarray`

Functions for processes and process groups `alarm`, `exec`, `fork`, `getpgrp`, `getppid`, `getpriority`, `kill`, `pipe`, `qx/STRING/`, `setpgrp`, `setpriority`, `sleep`, `system`, `times`, `wait`, `waitpid`

Keywords related to perl modules

`do`, `import`, `no`, `package`, `require`, `use`

Keywords related to classes and object-orientedness

`bless`, `dbmclose`, `dbmopen`, `package`, `ref`, `tie`, `tied`, `untie`, `use`

Low-level socket functions

`accept`, `bind`, `connect`, `getpeername`, `getsockname`, `getsockopt`, `listen`, `recv`, `send`, `setsockopt`, `shutdown`, `socket`, `socketpair`

System V interprocess communication functions

`msgctl`, `msgget`, `msgrcv`, `msgsnd`, `semctl`, `semget`, `semop`, `shmctl`, `shmget`, `shmread`, `shmwrite`

Fetching user and group info

`endgrent`, `endhostent`, `endnetent`, `endpwent`, `getgrent`, `getgrgid`, `getgrnam`, `getlogin`, `getpwent`, `getpwnam`, `getpwuid`, `setgrent`, `setpwent`

Fetching network info

`endprotoent`, `endservent`, `gethostbyaddr`, `gethostbyname`, `gethostent`, `getnetbyaddr`, `getnetbyname`, `getnetent`, `getprotobyname`, `getprotobynumber`, `getprotoent`, `getservbyname`, `getservbyport`, `getservent`, `sethostent`, `setnetent`, `setprotoent`, `setservent`

Time-related functions

`gmtime`, `localtime`, `time`, `times`

Functions new in perl5

`abs`, `bless`, `chomp`, `chr`, `exists`, `formline`, `glob`, `import`, `lc`, `lcfirst`, `map`, `my`, `no`, `our`, `prototype`, `qx`, `qw`, `readline`, `readpipe`, `ref`, `sub*`, `sysopen`, `tie`, `tied`, `uc`, `ucfirst`, `untie`, `use`

* - `sub` was a keyword in perl4, but in perl5 it is an operator, which can be used in expressions.

Functions obsoleted in perl5

`dbmclose`, `dbmopen`

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

A file test, where X is one of the letters listed below. This unary operator takes one argument, either a filename or a filehandle, and tests the associated file to see if something is true about it. If the argument is omitted, tests $_, except for `-t`, which tests STDIN. Unless otherwise documented, it returns `1` for true and `''` for false, or the undefined value if the file doesn't exist. Despite the funny names, precedence is the same as any other named unary operator, and the argument may be parenthesized like any other unary operator. The operator may be any of:

```
    -r File is readable by effective uid/gid.
    -w File is writable by effective uid/gid.
    -x File is executable by effective uid/gid.
    -o File is owned by effective uid.

    -R File is readable by real uid/gid.
    -W File is writable by real uid/gid.
    -X File is executable by real uid/gid.
    -O File is owned by real uid.

    -e File exists.
    -z File has zero size (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.
    -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 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
 #...
```

```
    }
```

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. Such reasons may be for example network filesystem access controls, ACLs (access control lists), read-only filesystems, and unrecognized executable formats.

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 the `use filetest 'access'` the above-mentioned filetests will test whether the permission can (not) be granted using the access() 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. Read the documentation for the `filetest` pragma for more information.

Note that `-s/a/b/` does not do a negated substitution. Saying `-exp($foo)` still works as expected, however--only single letters following a minus are interpreted as file tests.

The `-T` and `-B` switches work as follows. The first block or so of the file is examined for odd characters such as strange control codes or characters with the high bit set. If too many strange characters (>30%) are found, it's a `-B` file; otherwise it's a `-T` file. Also, any file containing null in the first block is considered a binary file. If `-T` or `-B` is used on a filehandle, the current IO buffer is examined rather than the first block. Both `-T` and `-B` return true on a null file, or a file at EOF when testing a filehandle. Because you have to read a file to do the `-T` test, on most occasions you want to use a `-f` against the file first, as in `next unless -f $file && -T $file`.

If any of the file tests (or either the `stat` or `lstat` operators) are given the special filehandle consisting of a solitary underline, then the stat structure of the previous file test (or stat operator) is used, saving a system call. (This doesn't work with `-t`, and you need to remember that lstat() and `-l` will leave values in the stat structure for the symbolic link, not the real file.) (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 _;
```

abs VALUE

abs

Returns the absolute value of its argument. If VALUE is omitted, uses $_.

accept NEWSOCKET,GENERICSOCKET

Accepts an incoming socket connect, just as the accept(2) system call does. Returns the packed address if it succeeded, false otherwise. See 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, you may 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. The Time::HiRes module (from CPAN, and starting from Perl 5.8 part of the standard distribution) may also prove useful.

It is usually a mistake to intermix `alarm` and `sleep` calls. (`sleep` may be internally implemented in 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*.

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])  }
```

Note that atan2(0, 0) is not well-defined.

bind SOCKET,NAME

Binds a network address to a socket, just as the bind system call does. Returns true if it succeeded, false otherwise. NAME should be a packed address of the appropriate type for the socket. See the examples in *"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 to always use it when appropriate, and to never use it when it isn't appropriate. Also, people can set their I/O to be by default UTF-8 encoded Unicode, not bytes.

In other words: regardless of platform, use binmode() on binary data, like for example images.

If LAYER is present it is a single string, but may contain multiple directives. The directives alter the behaviour of the file handle. When LAYER is present using binmode on 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) or elsewhere, `:raw` is *not* the simply inverse of `:crlf` -- other layers which would affect binary nature of the stream are *also* disabled. See *PerlIO*, *perlrun* and the discussion about the PERLIO environment variable.

The `:bytes`, `:crlf`, and `: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`.

In general, binmode() should be called after open() but before any I/O is done on the filehandle. Calling binmode() will normally flush 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 will operate on UTF-8 encoded Unicode characters.

The operating system, device drivers, C libraries, and Perl run-time system all work together to let the programmer treat a single character (`\n`) as the line terminator, irrespective of the 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.

Mac OS, all variants of Unix, 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 Mac OS and 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 contains `\cZ`, the I/O subsystem will regard it as the end of the file, unless you use binmode().

binmode() is not only important 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.

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 *perltoot* and *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*.

caller EXPR

caller

Returns the context of the current subroutine call. In scalar context, returns the caller's package name if there is a caller, that is, if we're in a subroutine or `eval` or `require`, and the undefined value otherwise. In list context, returns

```
    ($package, $filename, $line) = caller;
```

With EXPR, it returns some extra information that the debugger uses to print a stack trace. The value of EXPR indicates how many call frames to go back before the current one.

```
    ($package, $filename, $line, $subroutine, $hasargs,
    $wantarray, $evaltext, $is_require, $hints, $bitmask) =
caller($i);
```

Here $subroutine may be `(eval)` if the frame is not a subroutine call, but an `eval`. In such a case additional elements $evaltext and `$is_require` are set: `$is_require` is true if the frame is created by a `require` or `use` statement, $evaltext contains the text of the `eval EXPR` statement. In particular, for an `eval BLOCK` statement, $filename is `(eval)`, but $evaltext is undefined. (Note also that each `use` statement creates a `require` frame inside an `eval EXPR` frame.) $subroutine may also be `(unknown)` if this particular subroutine happens to have been deleted from the symbol table. `$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. The `$hints` and `$bitmask` values are subject to change between versions of Perl, and are not meant for external use.

Furthermore, when called from within the DB package, caller returns more detailed information: it sets the list variable `@DB::args` to be the arguments with which the subroutine was invoked.

Be aware that the optimizer might have optimized call frames away before `caller` had a chance to get the information. That means that `caller(N)` might not return information about the call frame you expect it do, for `N > 1`. In particular, `@DB::args` might have information from the previous time `caller` was called.

chdir EXPR

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 upon success, false otherwise. See the example under `die`.

On systems that support fchdir, you might pass a file handle or directory handle as argument. On systems that don't support fchdir, passing handles produces a fatal error at run time.

chmod LIST

Changes the permissions of a list of files. The first element of the list must be the numerical mode, which should probably be an octal number, and which definitely should *not* be a string of octal digits: `0644` is okay, `'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, you might pass file handles among the files. On systems that don't support fchmod, passing file handles produces a fatal error at run time.

```
    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 ':mode';

    chmod S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH, @executables;
    # This is identical to the chmod 0755 of the above example.
```

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.

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.

If the `encoding` pragma is in scope then the lengths returned are calculated from the length of `$/` in Unicode characters, which is not always the same as the length of `$/` in the native encoding.

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.

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, you might pass file handles among the files. On systems that don't support fchown, passing file handles produces a fatal error at run time.

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);
```

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. Note that characters from 128 to 255 (inclusive) are by default not encoded in UTF-8 Unicode for backward compatibility reasons (but see *encoding*).

If NUMBER is omitted, uses $_.

For the reverse, use *ord*.

Note that under the `bytes` pragma the NUMBER is masked to the low eight bits.

See *perlunicode* and *encoding* 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 $_.

close FILEHANDLE

close

Closes the file or pipe associated with the file handle, returning true only if IO buffers are successfully flushed and closes the system file descriptor. Closes the currently selected filehandle if the argument is omitted.

You don't have to close FILEHANDLE if you are immediately going to do another `open` on it, because `open` will close it for you. (See `open`.) However, an explicit `close` on an input file resets the line counter (`$.`), while the implicit close done by `open` does not.

If the file handle came from a piped open, `close` will additionally return false if one of the other system calls involved fails, or if the program exits with non-zero status. (If the only problem was that the program exited non-zero, `$!` will be set to `0`.) Closing a pipe also waits for the process executing on the pipe to complete, in case you want to look at the output of the pipe afterwards, and implicitly puts the exit status value of that command into $?.

Prematurely closing the read end of a pipe (i.e. before the process writing to it at the other end has closed it) will result in a SIGPIPE being delivered to the writer. 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.

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 as the connect system call does. Returns true if it succeeded, false otherwise. NAME should be a packed address of the appropriate type for the socket. See the examples in *"Sockets: Client/Server Communication" in perlipc*.

continue 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` will behave as if they had been executed within the main block. So will `next`, but since it will execute a `continue` block, it may be more entertaining.

```
    while (EXPR) {
 ### redo always comes here
 do_something;
    } continue {
 ### next always comes here
 do_something_else;
 # then back the top to re-check EXPR
    }
    ### last always comes here
```

Omitting the `continue` section is semantically equivalent to using an empty one, logically enough. In that case, `next` goes directly back to check the condition at the top of the loop.

cos EXPR

cos

Returns the cosine of EXPR (expressed in radians). If EXPR is omitted, takes 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 munitions).

crypt() is a one-way hash function. The PLAINTEXT and SALT is 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, do not assume anything about the returned string itself, or how many bytes in the digest 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 the digest string 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) 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`.

dbmclose HASH

[This function has been largely superseded by the `untie` function.]

Breaks the binding between a DBM file and a hash.

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`). If your system supports only the older DBM functions, you may perform only one `dbmopen` in your program. In older versions of Perl, if your system had neither DBM nor ndbm, calling `dbmopen` produced a fatal error; it now falls back to sdbm(3).

If you don't have write access to the DBM file, you can only read hash variables, not set them. If you want to test whether you can write, either use file tests or try setting a dummy hash entry inside an `eval`, which will trap the error.

Note that functions such as `keys` and `values` may return huge lists when used on large DBM files. You may prefer to use the `each` function to iterate over large DBM files. Example:

```
    # print out history file offsets
    dbmopen(%HIST,'/usr/lib/news/history',0666);
    while (($key,$val) = each %HIST) {
print $key, ' = ', unpack('L',$val), "\n";
    }
    dbmclose(%HIST);
```

See also *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: $!";
```

defined EXPR

defined

Returns a Boolean value telling whether EXPR has a value other than the undefined value `undef`. If EXPR is not present, `$_` will be checked.

Many operations return `undef` to indicate failure, end of file, system error, uninitialized variable, and other exceptional conditions. This function allows you to distinguish `undef` from other values. (A simple Boolean test will not distinguish among `undef`, zero, the empty string, and `"0"`, which are all equally false.) Note that since `undef` is a valid scalar, its presence doesn't *necessarily* indicate an exceptional condition: `pop` returns `undef` when its argument is an empty array, *or* when the element to return happens to be `undef`.

You may also use `defined(&func)` to check whether subroutine `&func` has ever been defined. The return value is unaffected by any forward declarations of `&func`. Note that a subroutine which 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 has 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 then are surprised to discover that the number `0` and `""` (the zero-length string) are, in fact, defined values. For example, if you say

```
    "ab" =~ /a(.*)b/;
```

The pattern match succeeds, and `$1` is defined, despite the fact that it matched "nothing". 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 you're questioning the integrity of what you're trying to do. At other times, a simple comparison to `0` or `""` is what you want.

See also *undef*, *exists*, *ref*.

delete EXPR

Given an expression that specifies a hash element, array element, hash slice, or array slice, deletes the specified element(s) from the hash or array. In the case of an array, if the array elements happen to be at the end, the size of the array will shrink to the highest element that tests true for exists() (or 0 if no such element exists).

Returns a list with the same number of elements as the number of elements for which deletion was attempted. Each element of that list consists of either the value of the element deleted, or the undefined value. In scalar context, this means that you get the value of the last element deleted (or the undefined value if that element did not exist).

```
    %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 bar baz)}; # @array  is
(undef,undef,33)
```

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.

Deleting an array element effectively returns that position of the array to its initial, uninitialized state. Subsequently testing for the same element with exists() will return false. Also, deleting array elements in the middle of an array will not shift the index of the elements after them down. Use splice() for that. See *exists*.

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 of these are slower than just assigning the empty list or undefining %HASH or @ARRAY:

```
    %HASH = ();  # completely empty %HASH
    undef %HASH; # forget %HASH ever existed

    @ARRAY = (); # completely empty @ARRAY
    undef @ARRAY; # forget @ARRAY ever existed
```

Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash element, array element, hash slice, or array slice lookup:

```
    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

Outside an `eval`, prints the value of LIST to `STDERR` and exits with the current value of `$!` (errno). If `$!` is `0`, exits with the value of `($? >> 8)` (backtick `command` status). If `($? >> 8)` is `0`, exits with `255`. Inside an `eval(),` the error message is stuffed into `$@` and the `eval` is terminated with the undefined value. This makes `die` the way to raise an exception.

Equivalent examples:

```
    die "Can't cd to spool: $!\n" unless chdir
'/usr/spool/news';
    chdir '/usr/spool/news' or die "Can't cd to spool: $!\n"
```

If the 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.
```

See also exit(), warn(), and the Carp module.

If LIST is empty and $@ already contains a value (typically from a previous eval) that value is reused after appending "\t...propagated". This is useful for propagating exceptions:

```
    eval { ... };
    die unless $@ =~ /Expected exception/;
```

If LIST 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.

die() can also be called with a reference argument. If this happens to be trapped within an eval(), $@ contains the reference. This behavior permits a more elaborate exception handling implementation using objects that maintain arbitrary state about the nature of the exception. Such a scheme is sometimes preferable to matching particular string values of $@ using regular expressions. Here's an example:

```
    use Scalar::Util 'blessed';

    eval { ... ; die Some::Module::Exception->new( FOO => "bar"
) };
    if ($@) {
        if (blessed($@) && $@->isa("Some::Module::Exception")) {
            # handle Some::Module::Exception
        }
        else {
            # handle all other possible exceptions
        }
    }
```

Because perl will stringify uncaught exception messages before displaying them, you may want to overload stringification operations on such custom 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 $SIG{__DIE__} hook. The associated handler will be called with the error text and can change the error message, if it sees fit, by calling die again. See *"$SIG{expr}" 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 the case--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.

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 SUBROUTINE(LIST)

This form of subroutine call is deprecated. See *perlsub*.

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 just like

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

except that it's more efficient and concise, keeps track of the current filename for error messages, searches the @INC directories, and updates `%INC` if the file is found. See *"Predefined Names" 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` cannot read the file, it returns undef and sets `$!` to the error. If `do` can read the file but cannot compile it, it returns undef and sets an error message in `$@`. If the file is successfully compiled, `do` returns the value of the last expression evaluated.

Note that inclusion of library modules is better done with the `use` and `require` operators, which also do automatic error checking and raise an exception if there's a problem.

You might like to use `do` to read in a program configuration file. Manual error checking can be done this way:

```
    # read in config files: system first, then user
    for $file ("/share/prog/defaults.rc",
               "$ENV{HOME}/.someprogrc")
   {
 unless ($return = do $file) {
     warn "couldn't parse $file: $@" if $@;
     warn "couldn't do $file: $!"    unless defined $return;
     warn "couldn't run $file"       unless $return;
 }
    }
```

dump LABEL

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.

WARNING: Any files opened at the time of the dump will *not* be open any more when the program is reincarnated, with possible resulting confusion on the part of Perl.

This function is now largely obsolete, partly because it's very hard to convert a core file into an executable, and because the real compiler backends for generating portable bytecode and compilable C code have superseded it. That's why you should now invoke it as `CORE::dump()`, if you don't want to be warned against a possible typo.

If you're looking to use *dump* to speed up your program, consider generating bytecode or native C code as described in *perlcc*. If you're just trying to accelerate a CGI script, consider using the `mod_perl` extension to **Apache**, or the CPAN module, CGI::Fast. You might also consider autoloading or selfloading, which at least make your program *appear* to run faster.

each HASH

When called in list context, returns a 2-element list consisting of the key and value for the next element of a hash, so that you can iterate over it. When called in scalar context, returns only the key for the next element in the hash.

Entries are returned in an apparently random order. The actual random order is subject to change in future versions of perl, but it is guaranteed to be in the same order as either the `keys` or `values` function would produce on the same (unmodified) hash. Since Perl 5.8.1 the ordering is different even between different runs of Perl for security reasons (see *"Algorithmic Complexity Attacks" in perlsec*).

When the hash is entirely read, a null array is returned in list context (which when assigned produces a false (`0`) value), and `undef` in scalar context. The next call to `each` after that will start iterating again. There is a single iterator for each hash, shared by all `each`, `keys`, and `values` function calls in the program; it can be reset by reading all the elements from the hash, or by evaluating `keys HASH` or `values HASH` . If you add or delete elements of a hash while you're iterating over it, you may get entries skipped or duplicated, so don't. Exception: It is always safe to delete the item most recently returned by `each()`, which means that the following code will work:

```
while (($key, $value) = each %hash) {
  print $key, "\n";
  delete $hash{$key};   # This is safe
}
```

The following prints out your environment like the printenv(1) program, only in a different order:

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

See also `keys`, `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 very useful in an interactive context.) Do not read from a terminal file (or call `eof(FILEHANDLE)` on it) after end-of-file is reached. 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 very 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, `eof()` will only detect the end of the last file. 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 if there was an error.

eval EXPR

eval BLOCK

eval

In the first form, the return value of EXPR is parsed and executed as if it were a little Perl program. The value of the expression (which is itself determined within scalar context) is first parsed, and if there weren't any errors, executed in the lexical context of the current Perl program, so that any variable settings or subroutine and format definitions remain afterwards. Note that the value is parsed every time the `eval` executes. If EXPR is omitted, evaluates $_. This form is typically used to delay parsing and subsequent execution of the text of EXPR until run time.

In the second form, the code within the BLOCK is parsed only once--at the same time the code surrounding the `eval` itself was parsed--and executed within the context of the current Perl program. This form is typically used to trap exceptions more efficiently than the first (see below), while also providing the benefit of checking the code within BLOCK at compile time.

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

In both forms, the value returned is the value of the last expression evaluated inside the mini-program; a return statement may be also used, just as with subroutines. The expression providing the return value is evaluated in void, scalar, or list context, depending on the context of the `eval` itself. See *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, an undefined value is returned by `eval`, and `$@` is set to the error message. If there was no error, `$@` is guaranteed to be a null string. Beware that using `eval` neither silences perl from printing warnings to STDERR, nor does it stuff the text of warning messages into `$@`. To do either of those, you have to use the `$SIG{__WARN__}` facility, or turn off warnings inside the BLOCK or EXPR using `no warnings 'all'`. See *warn*, *perlvar*, *warnings* and *perllexwarn*.

Note that, because `eval` traps otherwise-fatal errors, it is useful for determining whether a particular feature (such as `socket` or `symlink`) is implemented. It is also Perl's exception trapping mechanism, where the die operator is used to raise exceptions.

If the code to be executed doesn't vary, you may use the eval-BLOCK form to trap run-time errors without incurring the penalty of recompiling each time. The error, if any, is still returned in `$@`. Examples:

```
# make divide-by-zero nonfatal
eval { $answer = $a / $b; }; warn $@ if $@;

# same thing, but less efficient
eval '$answer = $a / $b'; warn $@ if $@;

# a compile-time error
eval { $answer = };   # WRONG

# a run-time error
eval '$answer ='; # sets $@
```

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 shown in this example:

```
# a very private exception trap for divide-by-zero
eval { local $SIG{'__DIE__'}; $answer = $a / $b; };
warn $@ if $@;
```

This is especially significant, given that `__DIE__` hooks can call `die` again, which has the effect of changing their error messages:

```
# __DIE__ hooks may modify error messages
{
   local $SIG{'__DIE__'} =
          sub { (my $x = $_[0]) =~ s/foo/bar/g; die $x };
   eval { die "foo lives here" };
   print $@ if $@;                # prints "bar lives here"
}
```

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.

`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.

Note that as a very special case, an `eval ''` executed within 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.

exec LIST

exec PROGRAM LIST

The `exec` function executes a system command *and never returns*-- use `system` instead of `exec` if you want it to return. It fails and returns false only if the command does not exist *and* it is executed directly instead of via your system's command shell (see below).

Since it's a common mistake to use `exec` instead of `system`, Perl warns you if there is a following statement which isn't `die`, `warn`, or `exit` (if `-w` is set - but you always do that). If you *really* want to follow an `exec` with some other statement, you can use one of these styles to avoid the warning:

```
    exec ('foo')   or print STDERR "couldn't exec foo: $!";
    { exec ('foo') }; print STDERR "couldn't exec foo: $!";
```

If there is more than one argument in LIST, or if LIST is an array with more than one value, calls execvp(3) with the arguments in LIST. If there is only one scalar argument or an array with one element in it, the argument is checked for shell metacharacters, and if there are any, the entire argument is passed to the system's command shell for parsing (this is `/bin/sh -c` on Unix platforms, but varies on other platforms). If there are no shell metacharacters in the argument, it is split into words and passed directly to `execvp`, which is more efficient. Examples:

```
    exec '/bin/echo', 'Your arguments are: ', @ARGV;
    exec "sort $outfile | uniq";
```

If you don't really want to execute the first argument, but want to lie to the program you are executing about its own name, you can specify the program you actually want to run as an "indirect object" (without a comma) in front of the LIST. (This always forces interpretation of the LIST as a multivalued list, even if there is only a single scalar in the list.) Example:

```
    $shell = '/bin/csh';
    exec $shell '-sh';  # pretend it's a login shell
```

or, more directly,

```
    exec {'/bin/csh'} '-sh'; # pretend it's a login shell
```

When the arguments get executed via the system shell, results will be subject to its quirks and capabilities. See *"`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 literally called *"echo surprise"*, didn't find it, and set $? to a non-zero value indicating failure.

Beginning with v5.6.0, Perl will attempt 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 in order to avoid lost output.

Note that `exec` will not call your `END` blocks, nor will it call any `DESTROY` methods in your objects.

exists EXPR

Given an expression that specifies a hash element or array element, returns true if the specified element in the hash or array has ever been initialized, even if the corresponding value is undefined. The element is not autovivified if it doesn't exist.

```
print "Exists\n"  if exists $hash{$key};
print "Defined\n"  if defined $hash{$key};
print "True\n"      if $hash{$key};

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 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 which 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 deepest nested array or hash 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:

```
    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.

See *"Pseudo-hashes: Using an array as a hash" in perlref* for specifics on how exists() acts when used on a pseudo-hash.

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. If this is a problem, you can call `POSIX:_exit($status)` to avoid END and destructor processing. See *perlmod* for details.

exp EXPR

exp

Returns *e* (the natural logarithm base) to the power of EXPR. If EXPR is omitted, gives `exp($_)`.

fcntl FILEHANDLE,FUNCTION,SCALAR

Implements the fcntl(2) function. You'll probably have to say

```
    use Fcntl;
```

first to get the correct constant definitions. Argument processing and value return works just like `ioctl` below. For example:

```
    use Fcntl;
    fcntl($filehandle, F_GETFL, $packed_return_buffer)
 or die "can't fcntl F_GETFL: $!";
```

You don't have to check for `defined` on the return from `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` will produce a fatal error 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";
```

fileno FILEHANDLE

Returns the file descriptor for a filehandle, or undefined if the filehandle is not open. 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) == fileno(THAT)) {
print "THIS and THAT are dups\n";
    }
```

(Filehandles connected to memory objects via new features of `open` may return undefined even though they are open.)

flock FILEHANDLE,OPERATION

Calls flock(2), or an emulation of it, on FILEHANDLE. Returns true for success, false on failure. Produces a fatal error if used on a machine that doesn't implement flock(2), fcntl(2) locking, or lockf(3). `flock` is Perl's portable file locking interface, although it locks only entire files, not records.

Two potentially non-obvious but traditional `flock` semantics are that it waits indefinitely until the lock is granted, and that its locks **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, or 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` will return immediately rather than blocking waiting for the lock (check the return status to see if you got it).

To avoid the possibility of 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 perl.

Here's a mailbox appender for BSD systems.

```
    use Fcntl ':flock'; # import LOCK_* constants

    sub lock {
 flock(MBOX,LOCK_EX);
 # and, in case someone appended
 # while we were waiting...
 seek(MBOX, 0, 2);
    }

    sub unlock {
 flock(MBOX,LOCK_UN);
    }

    open(MBOX, ">>/usr/spool/mail/$ENV{'USER'}")
     or die "Can't open mailbox: $!";

    lock();
    print MBOX $msg,"\n\n";
    unlock();
```

On systems that support a real flock(), locks are inherited across fork() calls, whereas those that must resort to the more capricious fcntl() function lose the locks, making it harder to write servers.

See also *DB_File* for other flock() examples.

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.

Beginning with v5.6.0, Perl will attempt 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 in order 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.

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 will treat the entire PICTURE as a single line. You may therefore need to use multiple formlines to implement a single record format, just like the format compiler.

Be careful if you put double quotes around the picture, because an `@` character may be taken to mean the beginning of an array name. `formline` always returns true. See *perlform* for other examples.

getc FILEHANDLE

getc

Returns the next character from the input file attached to FILEHANDLE, or the undefined value at end of file, or if there was an error (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 null
    }
    print "\n";
```

Determination of whether $BSD_STYLE should be set is left as an exercise to the reader.

The `POSIX::getattr` function can do this more portably on systems purporting POSIX compliance. See also the `Term::ReadKey` module from your nearest CPAN site; details on CPAN can be found on *"CPAN" in perlmodlib*.

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 null, use `getpwuid`.

```
$login = getlogin || getpwuid($<) || "Kilroy";
```

Do not consider `getlogin` for authentication: it is not as secure as `getpwuid`.

getpeername SOCKET

Returns the packed sockaddr address of other end of the SOCKET connection.

```
use Socket;
$hersockaddr    = getpeername(SOCK);
($port, $iaddr) = sockaddr_in($hersockaddr);
$herhostname    = gethostbyaddr($iaddr, AF_INET);
$herstraddr     = inet_ntoa($iaddr);
```

getpgrp PID

Returns the current process group for the specified PID. Use a PID of `0` to get the current process group for the current process. Will raise an exception if used on a machine that doesn't implement getpgrp(2). If PID is omitted, returns process group of current process. Note that the POSIX version of `getpgrp` does not accept a PID argument, so only `PID==0` is truly portable.

getppid

Returns the process id of the parent process.

Note for Linux users: on Linux, the C functions `getpid()` and `getppid()` return different values from different threads. In order to be portable, this behavior is not reflected by the perl-level function `getppid()`, that returns a consistent value across threads. If you want to call the underlying `getppid()`, you may use the CPAN module `Linux::Pid`.

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).

getpwnam NAME

getgrnam NAME

gethostbyname NAME

getnetbyname NAME

getprotobyname NAME

getpwuid UID

getgrgid GID

getservbyname NAME,PROTO

gethostbyaddr ADDR,ADDRTYPE

getnetbyaddr ADDR,ADDRTYPE

getprotobynumber NUMBER

getservbyport PORT,PROTO

getpwent

getgrent

gethostent

getnetent

getprotoent

getservent

setpwent

setgrent

sethostent STAYOPEN

setnetent STAYOPEN

setprotoent STAYOPEN

setservent STAYOPEN

endpwent

endgrent

endhostent

endnetent

endprotoent

endservent

These routines perform the same functions as their counterparts in the system library. In list context, the return values from the various get routines are as follows:

```
($name,$passwd,$uid,$gid,
   $quota,$comment,$gcos,$dir,$shell,$expire) = getpw*
($name,$passwd,$gid,$members) = getgr*
($name,$aliases,$addrtype,$length,@addrs) = gethost*
($name,$aliases,$addrtype,$net) = getnet*
($name,$aliases,$proto) = getproto*
($name,$aliases,$port,$proto) = getserv*
```

(If the entry doesn't exist you get a null list.)

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, because of 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 cases in the sense that in many systems they are unsupported. If the $quota is unsupported, it is an empty scalar. If it is supported, it usually encodes the disk quota. If the $comment field is unsupported, it is an empty scalar. If it is supported it usually encodes some administrative comment about the user. In some systems the $quota field may be $change or $age, fields that have to do with password aging. In some systems the $comment field may be $class. The $expire field, if present, encodes the expiration period of the account or the password. For the availability and the exact meaning of these fields in your system, please consult your getpwnam(3) documentation and your *pwd.h* file. You can also find out from within Perl 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 only supported 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 the raw addresses returned by the corresponding system library call. In the Internet domain, each address is four bytes long and you can unpack it by saying something like:

```
    ($a,$b,$c,$d) = unpack('C4',$addr[0]);
```

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);
```

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 like they're the same method calls (uid), they aren't, because a `File::stat` object is different from a `User::pwent` object.

getsockname SOCKET

Returns the packed sockaddr address of this end of the SOCKET connection, 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 call returns a packed string representing the requested socket option, or `undef` if there is an error (the error reason will be in $!). What exactly is in the packed string depends in the LEVEL and OPTNAME, consult your system documentation for details. A very common case however is that the option is an integer, in which case the result will be a packed integer which you can decode using unpack with the `i` (or `I`) format.

An example testing if Nagle's algorithm is turned on 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 "Could not query TCP_NODELAY socket option: $!";
    my $nodelay = unpack("I", $packed);
    print "Nagle's algorithm is turned ", $nodelay ? "off\n" :
"on\n";
```

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*.

Beginning with v5.6.0, this operator is implemented using the standard `File::Glob` extension. See *File::Glob* for details.

gmtime EXPR

gmtime

Converts a time as returned by the time function to an 9-element list with the time localized for the standard Greenwich time zone. Typically used as follows:

```
    #  0    1    2     3     4    5     6     7     8
    ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
         gmtime(time);
```

All 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 is the month itself, in the range `0..11` with 0 indicating January and 11 indicating December. $year is the number of years since 1900. That is, $year is `123` in year 2023. $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 always `0`.

Note that the $year element is *not* simply the last two digits of the year. If you assume it is then you create non-Y2K-compliant programs--and you wouldn't want to do that, would you?

The proper way to get a complete 4-digit year is simply:

```
$year += 1900;
```

And to get the last two digits of the year (e.g., '01' in 2001) do:

```
$year = sprintf("%02d", $year % 100);
```

If EXPR is omitted, `gmtime()` uses the current time (`gmtime(time)`).

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

```
    $now_string = gmtime;  # e.g., "Thu Oct 13 04:54:34 1994"
```

If you need local time instead of GMT use the *localtime* builtin. See also the `timegm` function provided by the `Time::Local` module, and the strftime(3) and mktime(3) functions available via the *POSIX* module.

This scalar value is **not** locale dependent (see *perllocale*), but is instead a Perl builtin. To get somewhat similar but locale dependent date strings, see the example in *localtime*.

See *"gmtime" in perlport* for portability concerns.

goto LABEL

goto EXPR

goto &NAME

The `goto-LABEL` form finds the statement labeled with LABEL and resumes execution there. It may not be used to go into any construct that requires initialization, such as a subroutine or a `foreach` loop. It also can't be used to go into a construct that is optimized away, or to get out of a block or subroutine given to `sort`. It can be used to go almost anywhere else within the dynamic scope, including out of subroutines, but it's usually better to use some other construct such as `last` or `die`. The author of Perl has never felt the need to use this form of `goto` (in Perl, that is--C is another matter). (The difference being 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 a label name, whose scope will be resolved dynamically. This allows for computed `goto`s per FORTRAN, but isn't necessarily recommended if you're optimizing for maintainability:

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

The `goto-&NAME` form is 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.

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*, or *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 `0` (or whatever you've set the `$[` variable to--but don't do that). If the substring is not found, `index` returns one less than the base, ordinarily `-1`.

int EXPR

int

Returns the integer portion of EXPR. If EXPR is omitted, uses `$_`. You should not use this 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 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.

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

Returns a list consisting of all the keys of the named hash. (In scalar context, returns the number of keys.)

The keys are returned in an apparently random order. The actual random order is subject to change in future versions of perl, but it is guaranteed to be the same order as either the `values` or `each` function produces (given that the hash has not been modified). Since Perl 5.8.1 the ordering is different even between different runs of Perl for security reasons (see *"Algorithmic Complexity Attacks" in perlsec*).

As a side effect, calling keys() resets the HASH's internal iterator (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;
    }
```

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).

See also `each`, `values` and `sort`.

kill SIGNAL, LIST

Sends a signal to a list of processes. Returns the number of processes successfully signaled (which is not necessarily the same as the number actually killed).

```
    $cnt = kill 1, $child1, $child2;
    kill 9, @goners;
```

If SIGNAL is zero, no signal is sent to the process. This is a useful way to check that a child process is alive and hasn't changed its UID. See *perlport* for notes on the portability of this construct.

Unlike in the shell, if SIGNAL is negative, it kills process groups instead of processes. (On System V, a negative *PROCESS* number will also kill process groups, but that's not portable.) That means you usually want to use positive not negative signals. You may also use a signal name in quotes.

See *"Signals" in perlipc* for more details.

last LABEL

last

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

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

`last` 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 `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.

lc EXPR

lc

Returns a lowercased version of EXPR. This is the internal function implementing the `\L` escape in double-quoted strings. Respects current LC_CTYPE locale if `use locale` in force. See *perllocale* and *perlunicode* for more details about locale and Unicode support.

If EXPR is omitted, uses $_.

lcfirst EXPR

lcfirst

Returns the value of EXPR with the first character lowercased. This is the internal function implementing the `\l` escape in double-quoted strings. Respects current LC_CTYPE locale if `use locale` in force. See *perllocale* and *perlunicode* for more details about locale and Unicode support.

If EXPR is omitted, uses $_.

length EXPR

length

Returns the length in *characters* of the value of EXPR. If EXPR is omitted, returns length of $_. Note that this 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.

Note the *characters*: if the EXPR is in Unicode, you will get the number of characters, not the number of bytes. To get the length in bytes, use `do { use bytes; length(EXPR) }`, see *bytes*.

link OLDFILE,NEWFILE

Creates a new filename linked to the old filename. Returns true for success, false otherwise.

listen SOCKET,QUEUESIZE

Does the same thing that the listen system call does. Returns true if it succeeded, false otherwise. See 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.

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` is the month itself, 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"
```

`$year` is the number of years since 1900, not just the last two digits of the year. That is, `$year` is `123` in year 2023. The proper way to get a complete 4-digit year is simply:

```
    $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 (`localtime(time)`).

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

```
    $now_string = localtime;  # e.g., "Thu Oct 13 04:54:34 1994"
```

This scalar value is **not** locale dependent but is a Perl builtin. For GMT instead of local time use the *gmtime* builtin. See also the `Time::Local` module (to convert the second, minutes, hours, ... 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.

See *"localtime" in perlport* for portability concerns.

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.

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. (However, if you've said `use threads`, lock() is always a keyword.) See *threads*.

log EXPR

log

Returns the natural logarithm (base *e*) of EXPR. If EXPR is omitted, returns 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 EXPR

lstat

Does the same thing as the `stat` function (including setting the special _ filehandle) but stats a symbolic link instead of the file the symbolic link points to. If symbolic links are unimplemented on your system, a normal `stat` is done. For much more detailed information, please see the documentation for *stat*.

If EXPR is omitted, stats $_.

m//

The match operator. See *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, @nums);
```

translates a list of numbers to the corresponding characters. And

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

is just a funny way to write

```
    %hash = ();
    foreach $_ (@array) {
$hash{getkey($_)} = $_;
    }
```

Note that $_ 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.

`{` 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 its dealing with based 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 `+` 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$_", 1) } @array  # this also works
    %hash = map {  lc($_), 1  } @array  # as does 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 , at
end
```

and you get list of anonymous hashes each with only 1 entry.

mkdir FILENAME,MASK

mkdir FILENAME

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). If omitted, MASK defaults to 0777.

In general, it is better to create directories with 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.

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

msgget KEY,FLAGS

Calls the System V IPC function msgget(2). Returns the message queue id, or the undefined value if there is an error. See also *"SysV IPC" in perlipc* and `IPC::SysV` and `IPC::Msg` documentation.

msgrcv ID,VAR,SIZE,TYPE,FLAGS

Calls the System V IPC function msgrcv to receive a message from message queue ID into variable VAR with a maximum message size of SIZE. Note that 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, or false if there is an error. See also *"SysV IPC" in perlipc*, `IPC::SysV`, and `IPC::SysV::Msg` documentation.

msgsnd ID,MSG,FLAGS

Calls the System V IPC function msgsnd to send the message MSG to the message queue ID. MSG must begin with the native long integer message type, and be followed by the length of the actual message, and finally the message itself. This kind of packing can be achieved with `pack("l! a*", $type, $message)`. Returns true if successful, or false if there is an error. See also `IPC::SysV` and `IPC::SysV::Msg` documentation.

my EXPR

my TYPE EXPR

my EXPR : ATTRS

my TYPE EXPR : ATTRS

A `my` declares the listed variables to be local (lexically) to the enclosing block, file, or `eval`. If more than one value is listed, the list must be placed in parentheses.

The exact semantics and interface of TYPE and ATTRS are still evolving. TYPE is currently bound to the use of `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*.

next LABEL

next

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

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

Note that if there were a `continue` block on the above, it would get executed even on discarded lines. If the LABEL is omitted, the command refers to the innermost enclosing loop.

`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.

no Module VERSION LIST

no Module VERSION

no Module LIST

no Module

See the `use` function, which `no` is the opposite of.

oct EXPR

oct

Interprets EXPR as an octal string and returns the corresponding value. (If EXPR happens to start off with `0x`, interprets it as a hex string. 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 the standard Perl or C 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():

```
    $perms = (stat("filename"))[2] & 07777;
    $oct_perms = sprintf "%lo", $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 will automatically convert strings into numbers as needed, this automatic conversion assumes base 10.)

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.

(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) the variable is assigned a reference to a new anonymous filehandle, otherwise if FILEHANDLE is an expression, its value is used as the name of the real filehandle wanted. (This is considered a symbolic reference, so `use strict 'refs'` should *not* be in effect.)

If EXPR is omitted, the scalar variable of the same name as the FILEHANDLE contains the filename. (Note that lexical variables--those declared with `my`--will not work for this purpose; so if you're using `my`, specify EXPR in your call to open.)

If three or more arguments are specified then the mode of opening and the file name are separate. If MODE is `'<'` or nothing, the file is opened for input. If MODE is `'>'`, the file is truncated and opened for output, being created if necessary. 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' `umask` value.

These various prefixes correspond to the fopen(3) modes of `'r'`, `'r+'`, `'w'`, `'w+'`, `'a'`, and `'a+'`.

In the 2-arguments (and 1-argument) form of the call the mode and filename should be concatenated (in this order), possibly separated by spaces. It is possible to omit the mode in these forms if the mode is `'<'`.

If the filename begins with `'|'`, the filename is interpreted as a command to which output is to be piped, and if the filename ends with a `'|'`, the filename is interpreted as a command which pipes output to us. 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.)

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 which pipes output to us. In the 2-arguments (and 1-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" in perlipc* for alternatives.)

In the three-or-more argument form of pipe opens, 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 specified. Experimental "layers" may give extra LIST arguments meaning.

In the 2-arguments (and 1-argument) form opening `'-'` opens STDIN and opening `'>-'` opens STDOUT.

You may use the three-argument form of open to specify IO "layers" (sometimes also referred to as "disciplines") to be applied to the handle that affect how the input and output are processed (see *open* and *PerlIO* for more details). For example

```
open(FH, "<:utf8", "file")
```

will open the UTF-8 encoded file containing Unicode characters, see *perluniintro*. Note that if layers are specified in the three-arg form then default layers stored in ${^OPEN} (see *perlvar*; usually set by the **open** pragma or the switch **-CioD**) are ignored.

Open returns nonzero upon success, the undefined value otherwise. If the `open` involved a pipe, the return value happens to be the pid of the subprocess.

If you're 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, which delimit lines with a single character, and which encode that character in C as `"\n"`, do not need `binmode`. The rest need it.

When opening a file, it's usually a bad idea to continue normal execution if the request failed, so `open` is frequently used in connection with `die`. Even if `die` won't do what you want (say, in a CGI script, where you want to make a nicely formatted error message (but there are modules that can help with that problem)) you should always check the return value from opening a file. The infrequent exception is when working with an unopened filehandle is actually what you want to do.

As a special case the 3-arg form with a read/write mode and the third argument being `undef`:

```
    open(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.

Since v5.8.0, perl has built using PerlIO by default. Unless you've changed this (i.e. Configure -Uuseperlio), you can open file handles to "in memory" files held in Perl scalars via:

```
    open($fh, '>', \$variable) || ..
```

Though if you try to re-open STDOUT or STDERR as an "in memory" file, you have to close it first:

```
    close STDOUT;
    open STDOUT, '>', \$variable or die "Can't open STDOUT: $!";
```

Examples:

```
    $ARTICLE = 100;
    open ARTICLE or die "Can't find article $ARTICLE: $!\n";
    while (<ARTICLE>) {...

    open(LOG, '>>/usr/spool/news/twitlog'); # (log is reserved)
    # if the open fails, output is discarded

    open(DBASE, '+<', 'dbase.mine')  # open for update
 or die "Can't open 'dbase.mine' for update: $!";

    open(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 end up 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 3-arg 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), and vice versa. But with `open(A, '>>&=B')` the filehandles will share the same file descriptor.

Note that if you are using Perls older than 5.8.0, Perl will be using the standard C libraries' 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 Perl has been compiled with PerlIO or not by running `perl -V` and looking for `useperlio=` line. If `useperlio` is `define`, you have PerlIO, otherwise you don't.

If you open a pipe on the command `'-'`, i.e., either `'|-'` or `'-|'` with 2-arguments (or 1-argument) form of open(), then there is an implicit fork done, and the return value of open is the pid of the child within the parent process, and `0` within the child process. (Use `defined($pid)` to determine whether the open was successful.) The filehandle behaves normally for the parent, but i/o to that filehandle is piped from/to the STDOUT/STDIN of the child process. In the child process the filehandle isn't opened--i/o happens from/to the new STDOUT or STDIN. Typically this is used like the normal piped open when you want to exercise more control over just how the pipe command gets executed, such as when you are running setuid, and don't want to have to scan shell commands for metacharacters. The following triples 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 example in each block shows the pipe as "list form", which is not yet supported on all platforms. A good rule of thumb is that if your platform has true `fork()` (in other words, if your platform is UNIX) you can use the list form.

See *"Safe Pipe Opens" in perlipc* for more examples of this.

Beginning with v5.6.0, 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, and returns the status value in $?.

The filename passed to 2-argument (or 1-argument) form of open() will have leading and trailing whitespace deleted, and the normal redirection characters honored. This property, known as "magic open", can often be used to good effect. A user could specify a filename of *"rsh cat file |"*, or you could change certain filenames as needed:

```
    $filename =~ s/(.*\.gz)\s*$/gzip -dc < $1|/;
    open(FH, $filename) or die "Can't open $filename: $!";
```

Use 3-argument form to open a file with arbitrary weird characters in it,

```
    open(FOO, '<', $file);
```

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

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

(this may not work on some bizarre filesystems). One should conscientiously choose between the *magic* and 3-arguments form of open():

```
    open IN, $ARGV[0];
```

will allow the user to specify an argument of the form `"rsh cat file |"`, but will not work on a filename which happens to have a trailing space, while

```
    open IN, '<', $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>;
```

Using the constructor from the `IO::Handle` package (or one of its subclasses, such as `IO::File` or `IO::Socket`), you can generate anonymous filehandles that have the scope of whatever variables hold references to them, and automatically close whenever and however you leave that scope:

```
    use IO::File;
    #...
    sub read_myfile_munged {
 my $ALL = shift;
 my $handle = new IO::File;
 open($handle, "myfile") or die "myfile: $!";
 $first = <$handle>
     or return ();     # Automatically closed here.
 mung $first or die "mung failed"; # Or here.
 return $first, <$handle> if $ALL; # Or here.
 $first;     # Or here.
    }
```

See *seek* for some details about mixing reading and writing.

opendir DIRHANDLE,EXPR

Opens a directory named EXPR for processing by `readdir`, `telldir`, `seekdir`, `rewinddir`, and `closedir`. Returns true if successful. 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. DIRHANDLEs have their own namespace separate from FILEHANDLEs.

ord EXPR

ord

Returns the numeric (the native 8-bit encoding, like ASCII or EBCDIC, or Unicode) value of the first character of EXPR. If EXPR is omitted, uses $_.

For the reverse, see *chr*. See *perlunicode* and *encoding* for more about Unicode.

our EXPR

our EXPR TYPE

our EXPR : ATTRS

our TYPE EXPR : ATTRS

`our` associates a simple name with a package variable in the current package for use within the current scope. When `use strict 'vars'` is in effect, `our` lets you use declared global variables without qualifying them with package names, within the lexical scope of the `our` declaration. In this way `our` differs from `use vars`, which is package scoped.

Unlike `my`, which both allocates storage for a variable and associates a simple name with that storage for use within the current scope, `our` associates a simple name with a package variable in the current package, for use within the current scope. In other words, `our` has the same scoping rules as `my`, but does not necessarily create a variable.

If more than one value is listed, the list must be placed in parentheses.

```
    our $foo;
    our($bar, $baz);
```

An `our` declaration declares a global 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 `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*.

The only currently recognized `our()` attribute is `unique` which indicates that a single copy of the global is to be used by all interpreters should the program happen to be running in a multi-interpreter environment. (The default behaviour would be for each interpreter to have its own copy of the global.) Examples:

```
    our @EXPORT : unique = qw(foo);
    our %EXPORT_TAGS : unique = (bar => [qw(aa bb cc)]);
    our $VERSION : unique = "1.00";
```

Note that this attribute also has the effect of making the global readonly when the first new interpreter is cloned (for example, when the first new thread is created).

Multi-interpreter environments can come to being either through the fork() emulation on Windows platforms, or by embedding perl in a multi-threaded application. The `unique` attribute does nothing in all other environments.

Warning: the current implementation of this attribute operates on the typeglob associated with the variable; this means that `our $x : unique` also has the effect of `our @x : unique; our %x : unique`. This may be subject to change.

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 a converted integer may be represented by a sequence of 4 bytes.

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 value.
    C An unsigned char value.  Only does bytes.  See U for
Unicode.

    s A signed short value.
    S An unsigned short value.
   (This 'short' is _exactly_ 16 bits, which may differ from
    what a local C compiler calls 'short'.  If you want
    native-length shorts, use the '!' suffix.)

    i A signed integer value.
    I An unsigned integer value.
   (This 'integer' is _at_least_ 32 bits wide.  Its exact
           size depends on what a local C compiler calls 'int',
           and may even be larger than the 'long' described in
```

```
                   the next item.)

            l A signed long value.
            L An unsigned long value.
           (This 'long' is _exactly_ 32 bits, which may differ from
            what a local C compiler calls 'long'.  If you want
            native-length longs, use the '!' suffix.)

            n An unsigned short in "network" (big-endian) order.
            N An unsigned long in "network" (big-endian) order.
            v An unsigned short in "VAX" (little-endian) order.
            V An unsigned long in "VAX" (little-endian) order.
           (These 'shorts' and 'longs' are _exactly_ 16 bits and
            _exactly_ 32 bits, respectively.)

            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.
                   Causes a fatal error otherwise.)

            j   A signed integer value (a Perl internal integer, IV).
            J   An unsigned integer value (a Perl internal unsigned
integer, UV).

            f A single-precision float in the native format.
            d A double-precision float in the native format.

            F A floating point value in the native native format
                   (a Perl internal floating point value, NV).
            D A long double-precision float in the native format.
           (Long doubles are available only if your system supports long
            double values _and_ if Perl has been compiled to support
those.
                   Causes a fatal error otherwise.)

            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 UTF-8 internally
 (or UTF-EBCDIC in EBCDIC platforms).

            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.
            X Back up a byte.
            @ Null fill to absolute position, counted from the start of
                the innermost ()-group.
            ( Start of a ()-group.
```

The following rules apply:

- Each letter may optionally be followed by a number giving a repeat count. With all types except a, A, Z, b, B, h, H, @, x, X and P the pack function will gobble up that many values from the LIST. A * for the repeat count means to use however many items are left, except for @, x , X, where it is equivalent to 0, and u, where it is equivalent to 1 (or 45, what is the same). A numeric repeat count may optionally be enclosed in brackets, as in `pack 'C[80]', @arr`.

  One can replace the numeric repeat count by a template enclosed in brackets; then the packed length of this template in bytes is used as a count. For example, `x[L]` skips a long (it skips the number of bytes in a long); the template `$t X[$t] $t` unpack()s twice what $t 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 has the maximal possible alignment.

  When used with Z, * results in the addition of a trailing null byte (so the packed result will be one longer than the byte `length` of the item).

  The repeat count for u is interpreted as the maximal number of bytes to encode per line of output, with 0 and 1 replaced by 45.

- 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 necessary. When unpacking, A strips trailing spaces and nulls, Z strips everything after the first null, and a returns data verbatim. When packing, a, and Z are equivalent.

  If the value-to-pack is too long, it is truncated. If 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 byte under all circumstances.

- Likewise, the b and B fields pack a string that many bits long. Each byte of the input field of pack() generates 1 bit of the result. Each result bit is based on the least-significant bit of the corresponding input byte, i.e., on `ord($byte)%2`. In particular, bytes `"0"` and `"1"` generate bits 0 and 1, as do bytes `"\0"` and `"\1"`.

  Starting from the beginning of the input string of pack(), each 8-tuple of bytes is converted to 1 byte of output. With format b the first byte of the 8-tuple determines the least-significant bit of a byte, and with format B it determines the most-significant bit of a byte.

  If the length of the input string is not exactly divisible by 8, the remainder is packed as if the input string were padded by null bytes at the end. Similarly, during unpack()ing the "extra" bits are ignored.

  If the input string of pack() is longer than needed, extra bytes are ignored. A * for the repeat count of pack() means to use all the bytes of the input field. On unpack()ing the bits are converted to a string of `"0"`s and `"1"`s.

- The h and H fields pack a string that many nybbles (4-bit groups, representable as hexadecimal digits, 0-9a-f) long.

  Each byte of the input field of pack() generates 4 bits of the result. For non-alphabetical bytes the result is based on the 4 least-significant bits of the input byte, i.e., on `ord($byte)%16`. In particular, bytes `"0"` and `"1"` generate nybbles 0 and 1, as do bytes `"\0"` and `"\1"`. For bytes `"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`. The result for bytes `"g".."z"` and `"G".."Z"` is not well-defined.

Starting from the beginning of the input string of pack(), each pair of bytes is converted to 1 byte of output. With format `h` the first byte of the pair determines the least-significant nybble of the output byte, and 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 byte at the end. Similarly, during unpack()ing the "extra" nybbles are ignored.

If the input string of pack() is longer than needed, extra bytes are ignored. A `*` for the repeat count of pack() means to use all the bytes of the input field. On unpack()ing the bits are converted to a string of hexadecimal digits.

- The `p` type packs a pointer to a null-terminated string. You are responsible for ensuring the string is not a temporary value (which can potentially get deallocated before you get around to using the packed result). The `P` type 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 for unpack().
- The `/` template character allows packing and unpacking of strings where the packed structure contains a byte count followed by the string itself. You write *length-item*`/`*string-item*.

  The *length-item* can be any `pack` template letter, and describes how the length value is packed. The ones 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`, the *string-item* must, at present, be `"A*"`, `"a*"` or `"Z*"`. For `unpack` the length of the string is obtained from the *length-item*, but if you put in the '*' it will be ignored. For all other codes, `unpack` applies the length value to the next item, which must not have a repeat count.

  ```
      unpack 'C/a', "\04Gurusamy";        gives 'Guru'
      unpack 'a3/A* A*', '007 Bond  J ';  gives ('
  Bond','J')
      pack 'n/a* w/a*','hello,','world';  gives
  "\000\006hello,\005world"
  ```

  The *length-item* is not returned explicitly from `unpack`.

  Adding a count to the *length-item* letter is unlikely to do anything useful, unless that letter is `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 immediately followed by a `!` suffix to signify native shorts or longs--as you can see from above for example a bare `l` does mean exactly 32 bits, the native `long` (as seen by the local C compiler) may be larger. This is an issue mainly in 64-bit platforms. You can see whether using `!` makes any difference by

  ```
   print length(pack("s")), " ", length(pack("s!")),
  "\n";
   print length(pack("l")), " ", length(pack("l!")),
  "\n";
  ```

`i!` and `I!` also work but only because of completeness; 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 via *Config*:

```
       use Config;
       print $Config{shortsize},    "\n";
       print $Config{intsize},      "\n";
       print $Config{longsize},     "\n";
       print $Config{longlongsize}, "\n";
```

(The `$Config{longlongsize}` will be undefined if your system does not support long longs.)

- The integer formats `s`, `S`, `i`, `I`, `l`, `L`, `j`, and `J` are inherently non-portable between processors and operating systems because they obey the 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, the Intel and VAX CPUs are little-endian, while everybody else, for example Motorola m68k/88k, PPC, Sparc, HP PA, Power, and Cray are big-endian. Alpha and MIPS can be either: Digital/Compaq used/uses them in little-endian mode; SGI/Cray uses them in big-endian mode.

  The names `big-endian' and `little-endian' are comic references to the classic "Gulliver's Travels" (via the paper "On Holy Wars and a Plea for Peace" by Danny Cohen, USC/ISI IEN 137, April 1, 1980) and the egg-eating habits of the Lilliputians.

  Some systems may have even weirder byte orders such as

  ```
    0x56 0x78 0x12 0x34
    0x34 0x12 0x78 0x56
  ```

  You can see your system's preference with

  ```
    print join(" ", map { sprintf "%#02x", $_ }

  unpack("C*",pack("L",0x12345678))), "\n";
  ```

  The byteorder on the platform where Perl was built is also available via *Config*:

  ```
   use Config;
   print $Config{byteorder}, "\n";
  ```

  Byteorders `'1234'` and `'12345678'` are little-endian, `'4321'` and `'87654321'` are big-endian.

  If you want portable packed integers use the formats `n`, `N`, `v`, and `V`, their byte endianness and size are known. See also *perlport*.

- Real numbers (floats and doubles) are in the native machine format only; due to the multiplicity of floating formats around, and the lack of a standard "network" representation, no facility for interchange has been made. This means that packed floating point data written on one machine may not be readable on another - even if both use IEEE floating point arithmetic (as the endian-ness of the memory representation is not part of the IEEE spec). See also *perlport*.

Note that Perl uses doubles internally for all numeric calculation, and converting from double into float and thence back to double again will lose precision (i.e., `unpack("f", pack("f", $foo))` will not in general equal $foo).

- If the pattern begins with a `U`, the resulting string will be treated as UTF-8-encoded Unicode. You can force UTF-8 encoding on in a string with an initial `U0`, and the bytes that follow will be interpreted as Unicode characters. If you don't want this to happen, you can begin your pattern with `C0` (or anything else) to force Perl not to UTF-8 encode your string, and then follow this with a `U*` somewhere in your pattern.

- You must yourself do any alignment or padding by inserting for example enough `'x'`es while packing. There is no way to pack() and unpack() could know where the bytes are going to or coming from. Therefore `pack` (and `unpack`) handle their output and input as flat sequences of bytes.

- A ()-group is a sub-TEMPLATE enclosed in parentheses. A group may take a repeat count, both as postfix, and for unpack() also via the `/` template character. Within each repetition of a group, positioning with `@` starts again at 0. Therefore, the result of

```
    pack( '@1A((@2A)@3A)', 'a', 'b', 'c' )
```

  is the string "\0a\0\0bc".

- `x` and `X` accept `!` modifier. In this case they act as alignment commands: they jump forward/back to the closest position aligned at a multiple of `count` bytes. For example, to pack() or unpack() C's `struct {char c; 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 on the double's size.

  For alignment commands `count` of 0 is equivalent to `count` of 1; both result in no-ops.

- A comment in a TEMPLATE starts with `#` and goes to the end of line. White space may be used to separate pack codes from each other, but a `!` modifier and a repeat count must follow immediately.

- If TEMPLATE requires more arguments to pack() than actually given, pack() assumes additional `""` arguments. If TEMPLATE requires fewer arguments to pack() than actually given, extra arguments are ignored.

Examples:

```
    $foo = pack("CCCC",65,66,67,68);
    # foo eq "ABCD"
    $foo = pack("C4",65,66,67,68);
    # same thing
    $foo = pack("U4",0x24b6,0x24b7,0x24b8,0x24b9);
    # same thing with Unicode circled letters

    $foo = pack("ccxxcc",65,66,67,68);
    # foo eq "AB\0\0CD"
    # note: the above examples featuring "C" and "c" are true
    # only on ASCII and ASCII-derived systems such as ISO Latin
1
    # and UTF-8.  In EBCDIC the first example would be
    # $foo = pack("CCCC",193,194,195,196);

    $foo = pack("s2",1,2);
    # "\1\0\2\0" on little-endian
    # "\0\1\0\2" on big-endian

    $foo = pack("a4","abcd","x","y","z");
    # "abcd"

    $foo = pack("aaaa","abcd","x","y","z");
    # "axyz"

    $foo = pack("a14","abcdefg");
    # "abcdefg\0\0\0\0\0\0\0"

    $foo = pack("i9pl", gmtime);
    # a real struct tm (on my system anyway)

    $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
```

The same template may generally also be used in unpack().

package NAMESPACE

package

Declares the compilation unit as being in the given namespace. The scope of the package declaration is from the declaration itself through the end of the enclosing block, file, or eval (the same as the `my` operator). All further unqualified dynamic identifiers will be in this namespace. A package statement affects only dynamic variables--including those you've used `local` on--but *not* lexical variables, which are created with `my`. Typically it would be the first declaration in a file to be included by the `require` or `use` operator. You can switch into a package in more than one place; it merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and filehandles in other packages by prefixing the identifier with the package name and a double colon: `$Package::Variable`. If the package name is null, the `main` package as assumed. That is, `$::sail` is equivalent to `$main::sail` (as well as to `$main'sail`, still seen in older code).

If NAMESPACE is omitted, then there is no current package, and all identifiers must be fully qualified or lexicals. However, you are strongly advised not to make use of this feature. Its use can cause unexpected behaviour, even crashing some versions of Perl. It is deprecated, and will be removed from a future release.

See *"Packages" in perlmod* for more information about packages, modules, and classes. See *perlsub* for other scoping issues.

pipe READHANDLE,WRITEHANDLE

Opens a pair of connected pipes like the corresponding system call. Note that if you set up a loop of piped processes, deadlock can occur unless you are very careful. In addition, note that Perl's pipes use IO buffering, so you may need to set `$|` to flush your WRITEHANDLE after each command, depending on the application.

See *IPC::Open2*, *IPC::Open3*, and *"Bidirectional Communication" in perlipc* for examples of such things.

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*.

pop ARRAY

pop

Pops and returns the last value of the array, shortening the array by one element. Has an effect similar to

```
    $ARRAY[$#ARRAY--]
```

If there are no elements in the array, returns the undefined value (although this may happen at other times as well). If ARRAY is omitted, pops the `@ARGV` array in the main program, and the `@_` array in subroutines, just like `shift`.

pos SCALAR

pos

Returns the offset of where the last `m//g` search left off for the variable 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 performed 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. 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 LIST

print

Prints a string or a list of strings. Returns true if successful. FILEHANDLE may be a scalar variable name, in which case the variable contains the name of or a reference to the filehandle, thus introducing one level of indirection. (NOTE: If FILEHANDLE is a variable and the next token is a term, it may be misinterpreted as an operator unless you interpose a + or put parentheses around the arguments.) If FILEHANDLE is omitted, prints by default to standard output (or to the last selected output channel--see *select*). If LIST is also omitted, prints `$_` to the currently selected output channel. To set the default output channel to something other than STDOUT use the select operation. The current value of `$,` (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, and any subroutine that you call will have one or more of its expressions evaluated in list context. Also be careful not to follow the print keyword with a left parenthesis unless you want the corresponding right parenthesis to terminate the arguments to the print--interpose a + or put parentheses around all the arguments.

Note that if you're storing FILEHANDLEs in an array, or if you're using any other expression more complex than a scalar variable to retrieve it, you will have to use a block returning the filehandle value instead:

```
    print { $files[$i] } "stuff\n";
    print { $OK ? STDOUT : STDERR } "stuff\n";
```

printf FILEHANDLE FORMAT, LIST

printf FORMAT, LIST

Equivalent to `print FILEHANDLE sprintf(FORMAT, LIST)`, except that `$\` (the output record separator) is not appended. The first argument of the list will be interpreted as the `printf` format. See `sprintf` for an explanation of the format argument. If `use locale` is in effect, the character used for the decimal point in formatted real numbers is affected by the LC_NUMERIC locale. See *perllocale*.

Don't fall into the trap of using a `printf` when a simple `print` would do. The `print` is more efficient and less error prone.

prototype FUNCTION

Returns the prototype of a function as a string (or `undef` if the function has no prototype). FUNCTION is a reference to, or the name of, the function whose prototype you want to retrieve.

If FUNCTION is a string starting with `CORE::`, the rest is taken as a name for Perl builtin. If the builtin is not *overridable* (such as `qw//`) or its arguments cannot be expressed by a prototype (such as `system`) returns `undef` because the builtin does not really behave like a Perl function. Otherwise, the string describing the equivalent prototype is returned.

push ARRAY,LIST ,

Treats ARRAY as a stack, and pushes the values of LIST onto the end of ARRAY. The length of ARRAY increases by the length of LIST. Has the same effect as

```
    for $value (LIST) {
$ARRAY[++$#ARRAY] = $value;
    }
```

but is more efficient. Returns the number of elements in the array following the completed `push`.

q/STRING/

qq/STRING/

qr/STRING/

qx/STRING/

qw/STRING/

Generalized quotes. See *"Regexp Quote-Like Operators" in perlop*.

quotemeta EXPR

quotemeta

Returns the value of EXPR with all non-"word" characters backslashed. (That is, all characters not matching `/[A-Za-z_0-9]/` will be preceded by a backslash in the returned string, regardless of any locale settings.) This is the internal function implementing the `\Q` escape in double-quoted strings.

If EXPR is omitted, uses `$_`.

rand EXPR

rand

Returns a random fractional number greater than or equal to `0` and less than the value of EXPR. (EXPR should be positive.) If EXPR is omitted, the value `1` is used. Currently EXPR with the value `0` is also special-cased as `1` - this has not been documented 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.)

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 actually implemented in terms of either Perl's or system's fread() call. 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 UTF-8 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 an undefined value in scalar context or a null list in list context.

If you're planning to filetest the return values out of a `readdir`, you'd better prepend the directory in question. Otherwise, because we didn't `chdir` there, it would have been testing the wrong file.

```
    opendir(DIR, $some_dir) || die "can't opendir $some_dir:
$!";
    @dots = grep { /^\./ && -f "$some_dir/$_" } readdir(DIR);
    closedir DIR;
```

readline EXPR

Reads from the filehandle whose typeglob is contained in EXPR. In scalar context, 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 however you may have defined it 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 takes the necessary steps to ensure that `readline` was successful.

```
    for (;;) {
        undef $!;
        unless (defined( $line = <> )) {
            die $! if $!;
            last; # reached EOF
        }
        # ...
    }
```

readlink EXPR

readlink

Returns the value of a symbolic link, if symbolic links are implemented. If not, gives a fatal error. If there is some system error, returns the undefined value and sets `$!` (errno). If EXPR is omitted, uses `$_`.

readpipe EXPR

EXPR is executed as a system command. The collected standard output of the command is returned. In scalar context, it comes back as a single (potentially multi-line) string. In list context, returns a list of lines (however you've defined lines with `$/` or `$INPUT_RECORD_SEPARATOR`). This is the internal function implementing the `qx/EXPR/` operator, but you can use it directly. The `qx/EXPR/` operator is discussed in more detail in *"I/O Operators" in perlop*.

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 `:utf8` I/O layer (see 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 read.

redo LABEL

redo

The `redo` command restarts the loop block without evaluating the conditional again. The `continue` block, if any, is not executed. If the LABEL is omitted, the command refers to the innermost enclosing loop. 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 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 `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.

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
```

If the referenced object has been blessed into a package, then that package name is returned instead. You can think of `ref` as a `typeof` operator.

```
    if (ref($r) eq "HASH") {
 print "r is a reference to a hash.\n";
    }
    unless (ref($r)) {
 print "r is not a reference at all.\n";
    }
```

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.

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). A fatal error is produced at run time 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`. Has semantics similar to the following subroutine:

```
    sub require {
       my ($filename) = @_;
       if (exists $INC{$filename}) {
           return 1 if $INC{$filename};
           die "Compilation failed in require";
       }
       my ($realfilename,$result);
       ITER: {
           foreach $prefix (@INC) {
               $realfilename = "$prefix/$filename";
               if (-f $realfilename) {
                   $INC{$filename} = $realfilename;
                   $result = do $realfilename;
                   last ITER;
               }
           }
           die "Can't find $filename in \@INC";
       }
       if ($@) {
           $INC{$filename} = undef;
           die $@;
       } elsif (!$result) {
```

```
                delete $INC{$filename};
                die "$filename did not return true value";
            } else {
                return $result;
            }
        }
```

Note that the file will not be included twice under the same specified name.

The file must return true as the last statement to indicate successful execution of any initialization code, so it's customary to end such a file with `1;` unless you're sure it'll return true otherwise. But it's better just to put the `1;`, in case you add more statements.

If EXPR is a bareword, the require assumes a "*.pm*" extension and replaces "*::*" with "*/*" in the filename for you, to make it easy to load standard modules. This form of loading of modules does not risk altering your namespace.

In other words, if you try this:

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

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

But if you try this:

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

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

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

Now that you understand how `require` looks for files in the case of 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 filename with a "*.pmc*" extension. A file with this extension is assumed to be Perl bytecode generated by *B::Bytecode*. If this file is found, and its modification time is newer than a coinciding " *.pm*" non-compiled file, it will be loaded in place of that non-compiled file ending in a " *.pm*" extension.

You can also insert hooks into the import facility, by putting directly Perl code 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 being a reference to itself, and the second the name of the file to be included (e.g. "*Foo/Bar.pm*"). The subroutine should return `undef` or a filehandle, from which the file to include will be read. If `undef` is returned, `require` will look at the remaining elements of @INC.

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 enables to pass indirectly some 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 it 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, new Foo(...);
```

Note that 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. Resets only variables or searches in the current package. Always returns 1. Examples:

```
    reset 'X';  # reset all X variables
    reset 'a-z'; # reset lower case variables
    reset;  # just reset ?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 a void context.

(Note that in the absence of an explicit `return`, a subroutine, eval, or do FILE will automatically return the value of the last expression evaluated.)

reverse LIST

In list context, returns a list value consisting of the elements of LIST in the opposite order. In scalar context, concatenates the elements of LIST and returns a string value with all characters in the opposite order.

```
    print reverse <>;  # line tac, last line first

    undef $/;   # for efficiency of <>
    print scalar reverse <>; # character tac, last line tsrif
```

Used without arguments in scalar context, reverse() reverses $_.

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.

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 $_.

s///

The substitution operator. See *perlop*.

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 unary operator, if you accidentally use for EXPR a parenthesized list, 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` upon success, `0` otherwise.

Note the *in bytes*: even if the filehandle has been set to operate on characters (for example by using the `: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 file for `sysread` or `syswrite`, don't use `seek`--buffering makes its effect on the file's system position unpredictable and non-portable. Use `sysseek` instead.

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 might have to stick in a seek() to reset things. The `seek` doesn't change the current position, but it *does* clear the end-of-file condition on the handle, so that the next `<FILE>` makes Perl try again to read something. We hope.

If that doesn't work (some IO implementations are particularly cantankerous), then you may need something more like this:

```
    for (;;) {
for ($curpos = tell(FILE); $_ = <FILE>;
            $curpos = tell(FILE)) {
    # search for some stuff and put it into files
}
sleep($for_a_while);
seek(FILE, $curpos, 0);
    }
```

seekdir DIRHANDLE,POS

Sets the current position for the `readdir` routine on DIRHANDLE. POS must be a value returned by `telldir`. `seekdir` also has the same caveats about possible directory compaction as the corresponding system library routine.

select FILEHANDLE

select

Returns the currently selected filehandle. Sets the current default filehandle for output, if FILEHANDLE is supplied. This has two effects: first, a `write` or a `print` without a filehandle will default to this FILEHANDLE. Second, references to variables related to output will refer to this output channel. For example, if you have to set the top of form format for more than one output channel, you might do the following:

```
    select(REPORT1);
    $^ = 'report1_top';
    select(REPORT2);
    $^ = 'report2_top';
```

FILEHANDLE may be an expression whose value gives the name of the actual filehandle. Thus:

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

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

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

select RBITS,WBITS,EBITS,TIMEOUT

This calls the select(2) system call with the bit masks specified, which can be constructed using `fileno` and `vec`, along these lines:

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

If you want to select on many filehandles you might wish to write a subroutine:

```
    sub fhbits {
 my(@fhlist) = split(' ',$_[0]);
 my($bits);
 for (@fhlist) {
     vec($bits,fileno($_),1) = 1;
 }
 $bits;
    }
    $rin = fhbits('STDIN TTY SOCK');
```

The usual idiom is:

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

or to block until something becomes ready just do this

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

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

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

You can effect a sleep of 250 milliseconds this way:

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

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 like the select(2) system call : it returns -1 and sets `$!`.

Note: on some Unixes, the select(2) system call may report a socket file descriptor as "ready for reading", when actually no data is available, thus a subsequent read blocks. It can be avoided using always the O_NONBLOCK flag on the socket. See select(2) and fcntl(2) for further details.

WARNING: One should not attempt to mix buffered I/O (like `read` or <FH>) with `select`, except as permitted by POSIX, and even then only on POSIX systems. You have to use `sysread` instead.

semctl ID,SEMNUM,CMD,ARG

Calls the System V IPC function `semctl`. You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If CMD is IPC_STAT or GETALL, then ARG must be a variable 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.

semget KEY,NSEMS,FLAGS

Calls the System V IPC function semget. Returns the semaphore id, or the undefined value if there is an error. See also *"SysV IPC" in perlipc*, `IPC::SysV`, `IPC::SysV::Semaphore` documentation.

semop KEY,OPSTRING

Calls the System V IPC function semop to perform 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, or false if there is an 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.

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 C `sendto`. Returns the number of characters sent, or the undefined value if there is an error. The C system call sendmsg(2) 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 `: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. Will produce a fatal error if 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()`.

setpriority WHICH,WHO,PRIORITY

Sets the current priority for a process, a process group, or a user. (See setpriority(2).) Will produce a fatal error if used on a machine that doesn't implement setpriority(2).

setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL

Sets the socket option requested. Returns undefined if there is an error. 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 the Nagle's algorithm for a socket:

```
use Socket qw(IPPROTO_TCP TCP_NODELAY);
setsockopt($socket, IPPROTO_TCP, TCP_NODELAY, 1);
```

shift ARRAY

shift

Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If there are no elements in the array, returns the undefined value. If ARRAY is omitted, shifts the `@_` array within the lexical scope of subroutines and formats, and the `@ARGV` array at file scopes or within the lexical scopes established by the `eval ''`, `BEGIN {}`, `INIT {}`, `CHECK {}`, and `END {}` constructs.

See also `unshift`, `push`, and `pop`. `shift` and `unshift` do the same thing to the left end of an array that `pop` and `push` do to the right end.

shmctl ID,CMD,ARG

Calls the System V IPC function shmctl. You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If CMD is `IPC_STAT`, then ARG must be a variable that will hold the returned `shmid_ds` structure. Returns like ioctl: the undefined value for error, "`0` but true" for zero, or the actual return value otherwise. See also *"SysV IPC" in perlipc* and `IPC::SysV` documentation.

shmget KEY,SIZE,FLAGS

Calls the System V IPC function shmget. Returns the shared memory segment id, or the undefined value if there is an error. See also *"SysV IPC" in perlipc* and `IPC::SysV` documentation.

shmread ID,VAR,POS,SIZE

shmwrite ID,STRING,POS,SIZE

Reads or writes the System V shared memory segment ID starting at position POS for size SIZE by attaching to it, copying in/out, and detaching from it. When reading, VAR must be a variable that will hold the data read. When writing, if STRING is too long, only SIZE bytes are used; if STRING is too short, nulls are written to fill out SIZE bytes. Return true if successful, or false if there is an error. shmread() taints the variable. See also *"SysV IPC" in perlipc*, `IPC::SysV` documentation, and the

`IPC::Shareable` module from CPAN.

shutdown SOCKET,HOW

Shuts down a socket connection in the manner indicated by HOW, which has the same interpretation as in the system call of the same name.

```
    shutdown(SOCKET, 0);    # I/we have stopped reading data
    shutdown(SOCKET, 1);    # I/we have stopped writing data
    shutdown(SOCKET, 2);    # I/we have stopped using this
socket
```

This is useful with sockets when you want to tell the other side you're done writing but not done reading, or vice versa. It's also a more insistent form of close because it also disables the file descriptor in any forked copies in other processes.

sin EXPR

sin

Returns the sine of EXPR (expressed in radians). If EXPR is omitted, returns sine of $_.

For the inverse sine operation, you may use the `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 EXPR seconds, or forever if no EXPR. May be interrupted if the process receives a signal such as `SIGALRM`. Returns the number of seconds actually slept. You probably cannot mix `alarm` and `sleep` calls, because `sleep` is often implemented using `alarm`.

On some older systems, it may sleep up to a full second less than what you requested, depending on how it counts seconds. Most modern systems always sleep the full amount. They may appear to sleep longer than that, however, because your process might not be scheduled right away in a busy multitasking system.

For delays of finer granularity than one second, you may use Perl's `syscall` interface to access setitimer(2) if your system supports it, or else see *select* above. The Time::HiRes module (from CPAN, and starting from Perl 5.8 part of the standard distribution) may also help.

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 system call 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 system call of the same name. If unimplemented, yields a fatal error. 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.

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, `sort`s in standard string comparison order. If SUBNAME is specified, it gives the name of a subroutine that returns an integer less than, equal to, or greater than `0`, depending on how the elements of the 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 counter-productive to declare $a and $b as lexicals.

In either case, the subroutine may not be recursive. 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` is in effect, `sort LIST` sorts LIST according to the current collation locale. See *perllocale*.

sort() returns aliases into the original list, much as a for loop's index variable aliases the list elements. That is, modifying an element of a list returned by sort() (for example, in a `foreach`, `map` or `grep`) actually modifies the element in the original list. This is usually something to be avoided when writing clear code.

Perl 5.6 and earlier used a quicksort algorithm to implement sort. That algorithm was not stable, and *could* go quadratic. (A *stable* sort preserves the input order of elements that compare equal. Although quicksort's run time is O(NlogN) when averaged over all arrays of length N, the time can be O(N**2), *quadratic* behavior, for some inputs.) In 5.7, the quicksort implementation was replaced with a stable mergesort algorithm whose worst-case behavior is O(NlogN). But benchmarks indicated that for some inputs, on some platforms, the original quicksort was faster. 5.8 has a sort pragma for limited control of the sort. Its rather blunt control of the underlying algorithm may not persist into future Perls, but the ability to characterize the input or output in implementation independent ways quite probably will. See *sort*.

Examples:

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

# same thing, but with explicit sort routine
```

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

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

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

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

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

   # 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

   @new = sort {
($b =~ /=(\d+)/)[0] <=> ($a =~ /=(\d+)/)[0]
      ||
            uc($a)  cmp  uc($b)
   } @old;

   # same thing, but much more efficiently;
   # we'll build auxiliary indices instead
   # for speed
   @nums = @caps = ();
   for (@old) {
push @nums, /=(\d+)/;
push @caps, uc($_);
   }

   @new = @old[ sort {
  $nums[$b] <=> $nums[$a]
    ||
   $caps[$a] cmp $caps[$b]
         } 0..$#old
        ];

    # same thing, but without any temps
    @new = map { $_->[0] }
           sort { $b->[1] <=> $a->[1]
                           ||
                  $a->[2] cmp $b->[2]
           } map { [$_, /=(\d+)/, uc($_)] } @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;
```

If you're using strict, you *must not* declare $a and $b as lexicals. They are package globals. That means if you're in the `main` package 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), and because `sort` will trigger a fatal error unless the result of a comparison is defined, when sorting with a comparison function like `$a <=> $b`, be careful about lists that might contain a `NaN`. The following example takes advantage of the fact that `NaN != NaN` to eliminate any `NaN`s from the input.

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

splice ARRAY,OFFSET,LENGTH,LIST

splice ARRAY,OFFSET,LENGTH

splice ARRAY,OFFSET

splice ARRAY

Removes the elements designated by OFFSET and LENGTH from an array, and replaces them with the elements of LIST, if any. In list context, returns the elements removed from the array. In scalar context, returns the last element removed, or `undef` if no elements are removed. The array grows or shrinks as necessary. If OFFSET is negative then it starts that far from the end of the array. If LENGTH is omitted, removes everything from OFFSET onward. If LENGTH is negative, removes the elements from OFFSET onward except for -LENGTH elements at the end of the array. If both OFFSET and LENGTH are omitted, removes everything. If OFFSET is past the end of the array, perl issues a warning, and splices at the end of the array.

The following equivalences hold (assuming `$[ == 0 and $#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)
```

Example, assuming array lengths are passed before arrays:

```
    sub aeq { # compare two list values
 my(@a) = splice(@_,0,shift);
 my(@b) = splice(@_,0,shift);
 return 0 unless @a == @b; # same len?
 while (@a) {
     return 0 if pop(@a) ne pop(@b);
 }
 return 1;
    }
    if (&aeq($len,@foo[1..$len],0+@bar,@bar)) { ... }
```

split /PATTERN/,EXPR,LIMIT

split /PATTERN/,EXPR

split /PATTERN/

split

Splits the string EXPR into a list of strings and returns that list. By default, empty leading fields are preserved, and empty trailing ones are deleted. (If all fields are empty, they are considered to be trailing.)

In scalar context, returns the number of fields found and splits into the @_ array. Use of split in scalar context is deprecated, however, because it clobbers your subroutine arguments.

If EXPR is omitted, splits the $_ string. If PATTERN is also omitted, splits on whitespace (after skipping any leading whitespace). Anything matching PATTERN is taken to be a delimiter separating the fields. (Note that the delimiter may be longer than one character.)

If LIMIT is specified and positive, it represents the maximum number of fields the EXPR will be split into, though the actual number of fields returned depends on the number of times PATTERN matches within EXPR. If LIMIT is unspecified or zero, trailing null fields are stripped (which potential users of pop would do well to remember). If LIMIT is negative, it is treated as if an arbitrarily large LIMIT had been specified. Note that splitting an EXPR that evaluates to the empty string always returns the empty list, regardless of the LIMIT specified.

A pattern matching the null string (not to be confused with a null pattern `//`, which is just one member of the set of patterns matching a null string) will split the value of EXPR into separate characters at each point it matches that way. For example:

```
    print join(':', split(/ */, 'hi there'));
```

produces the output 'h:i:t:h:e:r:e'.

As a special case for `split`, using the empty pattern `//` specifically matches only the null string, and is not be confused with the regular use of `//` to mean "the last successful pattern match". So, for `split`, the following:

```
    print join(':', split(//, 'hi there'));
```

produces the output 'h:i: :t:h:e:r:e'.

Empty leading (or trailing) fields are produced when there are positive width matches at the beginning (or end) of the string; a zero-width match at the beginning (or end) of the string does not produce an empty field. For example:

```
   print join(':', split(/(?=\w)/, 'hi there!'));
```

produces the output 'h:i :t:h:e:r:e!'.

The LIMIT parameter can be used to split a line partially

```
    ($login, $passwd, $remainder) = split(/:/, $_, 3);
```

When assigning to a list, if LIMIT is omitted, or zero, Perl supplies a LIMIT one larger than the number of variables in the list, to avoid unnecessary work. For the list above LIMIT would have been 4 by default. In time critical applications it behooves you not to split into more fields than you really need.

If the PATTERN contains parentheses, additional list elements are created from each matching substring in the delimiter.

```
    split(/([,-])/, "1-10,20", 3);
```

produces the list value

```
    (1, '-', 10, ',', 20)
```

If you had the entire header of a normal Unix email message in $header, you could split it up into fields and their values this way:

```
    $header =~ s/\n\s+/ /g;  # fix continuation lines
    %hdrs   =  (UNIX_FROM => split /^(\S*?):\s*/m, $header);
```

The pattern `/PATTERN/` may be replaced with an expression to specify patterns that vary at runtime. (To do runtime compilation only once, use `/$variable/o`.)

As a special case, specifying a PATTERN of space (`' '`) will split on white space just as `split` with no arguments does. Thus, `split(' ')` can be used to emulate **awk**'s default behavior, whereas `split(/ /)` will give you as many null initial fields as there are leading spaces. A `split` on `/\s+/` is like a `split(' ')` except that any leading whitespace produces a null first field. A `split` with no arguments really does a `split(' ', $_)` internally.

A PATTERN of `/^/` is treated as if it were `/^/m`, since it isn't much use otherwise.

Example:

```
    open(PASSWD, '/etc/passwd');
    while (<PASSWD>) {
        chomp;
        ($login, $passwd, $uid, $gid,
         $gcos, $home, $shell) = split(/:/);
 #...
    }
```

As with regular pattern matching, any capturing parentheses that are not matched in a `split()` will be set to `undef` when returned:

```
    @fields = split /(A)|B/, "1A2B3";
    # @fields is (1, 'A', 2, undef, 3)
```

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`, but it doesn't use it (except for floating-point numbers, and even then only the standard modifiers are allowed). As a result, any non-standard extensions in your local `sprintf` are not available from Perl.

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
   %p a pointer (outputs the Perl value's address in
hexadecimal)
   %n special: *stores* the number of characters output so far
        into the next variable in the parameter list
```

Finally, for backward (and we do mean "backward") compatibility, Perl permits these unnecessary but widely-supported conversions:

```
   %i a synonym for %d
   %D a synonym for %ld
   %U a synonym for %lu
   %O a synonym for %lo
   %F a synonym for %f
```

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".

Between the % and the format letter, you may specify a number of 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, e.g.:

```
  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 positive number with a space + prefix positive number with a plus sign - left-justify within the field 0 use zeros, not spaces, to right-justify # prefix non-zero octal with "0", non-zero hex with "0x", non-zero binary with "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 '<%#x>', 12;   # prints "<0xc>"
```

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 e.g. *2$v:

```
  printf '%*4$vX %*4$vX %*4$vX', @addr[1..3], ":";   # 3
IPv6 addresses
```

(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 (with e.g. *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, with the exception of 'g' and 'G', this specifies the number of decimal places to show (the default being 6), e.g.:

```
  # 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 prior to the decimal point as well as after it, e.g.:

```
  # 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:

```
  printf '<%.6x>', 1;      # prints "<000001>"
  printf '<%#.6x>', 1;     # prints "<0x000001>"
  printf '<%-10.6x>', 1;   # prints "<000001    >"
```

For string conversions, specifying a precision truncates the string to fit in 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>"
```

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

```
   l           interpret integer as C type "long" or
"unsigned long"
   h           interpret integer as C type "short" or
"unsigned short"
   q, L or ll  interpret integer as C type "long long",
"unsigned long long".
               or "quads" (typically 64-bit integers)
```

The last will produce errors if Perl does not understand "quads" in your installation. (This requires that either the platform natively supports quads or Perl was specifically compiled to support quads.) You can find out whether your Perl supports quads via *Config*:

```
use Config;
($Config{use64bitint} eq 'define' || $Config{longsize} >=
8) &&
 print "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;
$Config{d_longdbl} eq 'define' && print "long doubles\n";
```

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;
        ($Config{uselongdouble} eq 'define') &&
                print "long doubles by default\n";
```

It can also be the case 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 it is supported for compatibility with XS code; it means 'use the standard size for a Perl integer (or floating-point number)', which is already the default for Perl code.

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 in which they appear in the format specification *before* the value to format. Where an argument is specified using 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 in any case).

So:

```
  printf '<%*.*s>', $a, $b, $c;
```

would use $a for the width, $b for the precision and $c as the value to format, while:

```
  print '<%*1$.*s>', $a, $b;
```

would use $a for the width and the precision, and $b as the value to format.

Here are some more examples - beware that when using an explicit index, the $ may need to be escaped:

```
  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` is in effect, the character used for the decimal point in formatted real numbers is affected by the LC_NUMERIC locale. See *perllocale*.

sqrt EXPR

sqrt

Return the square root of EXPR. If EXPR is omitted, returns square root of $_. Only works on non-negative operands, unless you've loaded the standard Math::Complex module.

```
    use Math::Complex;
    print sqrt(-2);    # prints 1.4142135623731i
```

srand EXPR

srand

Sets 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.

If srand() is not called explicitly, it is called implicitly at the first use of the `rand` operator. However, this was not the case in versions of Perl before 5.004, so if your script will run under older Perl versions, it should call `srand`.

Most programs won't even call srand() at all, except those that need a cryptographically-strong starting point rather than the generally acceptable default, which is based on time of day, process ID, and memory allocation, or the */dev/urandom* device, if available.

You can call srand($seed) with the same $seed to reproduce the *same* sequence from rand(), but this is usually reserved for generating predictable results for testing or debugging. Otherwise, don't call srand() more than once in your program.

Do **not** call srand() (i.e. without an argument) more than once in a script. 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.

In versions of Perl prior to 5.004 the default seed was just the current `time`. This isn't a particularly good seed, so many old programs supply their own seed value (often `time ^ $$` or `time ^ ($$ + ($$ << 15))`, but that isn't necessary any more.

For cryptographic purposes, however, you need something much more random than the default seed. Checksumming the compressed output of one or more rapidly changing operating system status programs is the usual method. For example:

```
    srand (time ^ $$ ^ unpack "%L*", `ps axww | gzip`);
```

If you're particularly concerned with this, see the `Math::TrulyRandom` module in CPAN.

Frequently called programs (like CGI scripts) that simply use

```
    time ^ $$
```

for a seed can fall prey to the mathematical property that

```
    a^b == (a+1)^(b+1)
```

one-third of the time. So don't do that.

stat FILEHANDLE

stat EXPR

stat

Returns a 13-element list giving the status info for a file, either the file opened via FILEHANDLE, or named by EXPR. If EXPR is omitted, it stats $_. Returns a null list if the stat fails. Typically used as follows:

```
    ($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
       $atime,$mtime,$ctime,$blksize,$blocks)
           = stat($filename);
```

Not all fields are supported on all filesystem types. Here are the 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 block size for file system I/O
 12 blocks   actual number of blocks allocated
```

(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;
    $is_setgid     =  S_ISDIR($mode);
```

You could write the last two using the `-u` and `-d` operators. The 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 necessarily all are 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, 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 e.g. 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.

study SCALAR

study

Takes extra time to study SCALAR ($_ if unspecified) in anticipation of doing many pattern matches on the string before it is next modified. This may or may not save time, depending on the nature and number of patterns you are searching on, and on the distribution of character frequencies in the string to be searched--you probably want to compare run times with and without it to see which runs faster. Those loops that scan for many short constant strings (including the constant parts of more complex patterns) will benefit most. You may have only one `study` active at a time--if you study a different scalar the first is "unstudied". (The way `study` works is this: a linked list of every character in the string to be searched is made, so we know, for example, where all the `'k'` characters are. From each search string, the rarest character is selected, based on some static frequency tables constructed from some C programs and English text. Only those places that contain this "rarest" character are examined.)

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

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

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

Note that if you have to look for strings that you don't know till runtime, you can build an entire loop as a string and `eval` that to avoid recompiling all your patterns all the time. Together with undefining `$/` to input entire files as one record, this can be very fast, often faster than specialized programs like fgrep(1). The following scans a list of files (`@files`) for a list of words (`@words`), and prints out the names of those files that contain a match:

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

sub 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, and does actually return a value: the CODE ref of the closure you just created.

See *perlsub* and *perlref* for details about subroutines and references, and *attributes* and *Attribute::Handlers* for more information about attributes.

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 `0`, or whatever you've set `$[` to (but don't do that). If OFFSET is negative (or more precisely, less than `$[`), starts that far from the end of the string. If LENGTH is omitted, returns everything to the end of the string. If LENGTH is negative, leaves that many characters off the end of the string.

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 is a fatal error. 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';  # fatal error
```

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().

symlink OLDFILE,NEWFILE

Creates a new filename symbolically linked to the old filename. Returns `1` for success, `0` otherwise. On systems that don't support symbolic links, produces a fatal error at run time. To check for that, use eval:

```
    $symlink_exists = eval { symlink("",""); 1 };
```

syscall 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, produces a fatal error. The arguments are interpreted as follows: if a given argument is numeric, the argument is passed as an int. If not, the pointer to the string value is passed. You are responsible to make sure a string is pre-extended long enough to receive any result that might be written into a string. You can't use a string literal (or other read-only string) as an argument to `syscall` because Perl has to assume that any string pointer might be written through. If your integer arguments are not literals and have never been interpreted in a numeric context, you may need to add `0` to them to force them to look like numbers. This emulates the `syswrite` function (or vice versa):

```
    require 'syscall.ph';  # may need to run h2ph
    $s = "hi there\n";
    syscall(&SYS_write, fileno(STDOUT), $s, length $s);
```

Note that Perl supports passing of up to only 14 arguments to your system call, which in practice should usually suffice.

Syscall returns whatever value returned by the system call it calls. If the system call fails, `syscall` returns `-1` and sets `$!` (errno). Note that some system calls can legitimately return `-1`. The proper way to handle such calls is to assign `$!=0;` before the call and check the value of `$!` if syscall returns `-1`.

There's a problem with `syscall(&SYS_pipe)`: it returns the file number of the read end of the pipe it creates. There is no way to retrieve the file number of the other end. You can avoid this problem by using `pipe` instead.

sysopen FILEHANDLE,FILENAME,MODE

sysopen FILEHANDLE,FILENAME,MODE,PERMS

Opens the file whose filename is given by FILENAME, and associates it with FILEHANDLE. If FILEHANDLE is an expression, its value is used as the name of the real filehandle wanted. This function calls the underlying operating system's `open` function with the parameters FILENAME, MODE, PERMS.

The possible values and flag bits of the MODE parameter are system-dependent; they are available via the standard module `Fcntl`. See the documentation of your operating system's `open` 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: zero means read-only, one means write-only, and two means read/write. We know that these values do *not* work under OS/390 & VM/ESA Unix 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 rebuilding Perl to use the `sfio` library, or perhaps using the POSIX::open() function.

See *perlopentut* for a kinder, gentler explanation of opening files.

sysread FILEHANDLE,SCALAR,LENGTH,OFFSET

sysread FILEHANDLE,SCALAR,LENGTH

Attempts to read LENGTH bytes of data into variable SCALAR from the specified FILEHANDLE, using the system call read(2). It bypasses 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 very 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 the system call 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 `:utf8` I/O layer), tell() will return byte offsets, not character offsets (because implementing that would render sysseek() very slow).

sysseek() bypasses normal buffered IO, so mixing this 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 complete. Note that argument processing varies depending on the number of arguments. If there is more than one argument in LIST, or if LIST is an array with more than one value, starts the program given by the first element of the list with arguments given by the rest of the list. If there is only one scalar argument, the argument is checked for shell metacharacters, and if there are any, the entire argument is passed to the system's command shell for parsing (this is `/bin/sh -c` on Unix platforms, but varies on other platforms). If there are no shell metacharacters in the argument, it is split into words and passed directly to `execvp`, which is more efficient.

Beginning with v5.6.0, 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).

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: $?"
```

You can check all the failure possibilities 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;
   }
```

or more portably by using the W*() calls of the POSIX extension; see *perlport* for more information.

When the arguments get executed via the system shell, results and return codes will be subject to its quirks and capabilities. See *"`STRING`" in perlop* and *exec* for details.

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 the system call 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 buffers 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 available data in the SCALAR after the OFFSET, only as much data as is available will be written.

An OFFSET may be specified to write the data from some part of the string other than the beginning. A negative OFFSET specifies writing that many characters counting backwards from the end of the string. In the case the SCALAR is empty you can use OFFSET but only zero offset.

Note that if the filehandle has been marked as `:utf8`, Unicode characters are written instead of bytes (the LENGTH, OFFSET, and the return value of syswrite() are in UTF-8 encoded Unicode characters). The `:encoding(...)` layer implicitly introduces the `:utf8` layer. 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 `: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 file handle 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 `new` 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 `new` method is also returned by the `tie` function, which would be useful if you want to access other methods in CLASSNAME.

Note that functions such as `keys` and `values` may return huge lists when used on large objects, like DBM files. You may prefer to use the `each` function to iterate over such. Example:

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

A class implementing a hash should have the following methods:

```
    TIEHASH classname, LIST
    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
    DESTROY this
    UNTIE this
```

A class implementing a file handle 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, you may use either the Time::HiRes module (from CPAN, and starting from Perl 5.8 part of the standard distribution), or if you have gettimeofday(2), you may be able to use the `syscall` interface of Perl. See *perlfaq8* for details.

times

Returns a four-element list giving the user and system times, in seconds, for this process and the children of this process.

```
    ($user,$system,$cuser,$csystem) = times;
```

In scalar context, `times` returns `$user`.

tr///

The transliteration operator. Same as `y///`. See *perlop*.

truncate FILEHANDLE,LENGTH

truncate EXPR,LENGTH

Truncates the file opened on FILEHANDLE, or named by EXPR, to the specified length. Produces a fatal error if truncate isn't implemented on your system. Returns true if successful, the undefined value otherwise.

The behavior is undefined if LENGTH is greater than the length of the file.

uc EXPR

uc

Returns an uppercased version of EXPR. This is the internal function implementing the `\U` escape in double-quoted strings. Respects current LC_CTYPE locale if `use locale` in force. See *perllocale* and *perlunicode* for more details about locale and Unicode support. It does not attempt to do titlecase mapping on initial letters. See `ucfirst` for that.

If EXPR is omitted, uses `$_`.

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. Respects current LC_CTYPE locale if `use locale` in force. See *perllocale* and *perlunicode* for more details about locale and Unicode support.

If EXPR is omitted, uses `$_`.

umask EXPR

umask

Sets the umask for the process to EXPR and returns the previous value. If EXPR is omitted, merely returns the current umask.

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` (`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), produces a fatal error at run time. If umask(2) is not implemented and you are not trying to restrict access for yourself, returns `undef`.

Remember that a umask is a number, usually given in octal; it is *not* a string of octal digits. See also *oct*, if all you have is a string.

undef EXPR

undef

Undefines the value of EXPR, which must be an lvalue. Use only on a scalar value, an array (using `@`), a hash (using `%`), a subroutine (using `&`), or a typeglob (using `*`). (Saying `undef $hash{$key}` will probably not do what you expect on most predefined variables or DBM list values, so don't do that; see *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. Returns the number of files successfully deleted.

```
    $cnt = unlink 'a', 'b', 'c';
    unlink @goners;
    unlink <*.bak>;
```

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, uses $_.

unpack TEMPLATE,EXPR

`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.)

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 bytes 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("c",$_[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("%32C*",<>) % 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: in some cases, the repeat count is decreased, or `unpack()` will produce null strings or zeroes, or terminate with an error. If the input string is longer than one described by the TEMPLATE, the rest is ignored.

See *pack* for more examples and notes.

untie VARIABLE

Breaks the binding between a variable and a package. (See `tie`.) Has no effect if the variable is not tied.

unshift ARRAY,LIST

Does the opposite of a `shift`. Or the opposite of a `push`, depending on how you look at it. Prepends list to the front of the array, and returns the new number of elements in the array.

```
    unshift(@ARGV, '-e') unless $ARGV[0] =~ /^-/;
```

Note the LIST is prepended whole, not one element at a time, so the prepended elements stay in the same order. Use `reverse` to do the reverse.

use Module 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; import Module LIST; }
```

except that Module *must* be a bareword.

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. A fatal error is produced 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.

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.

```
    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 have changed in incompatible ways from older versions of Perl. (We try not to do this more than we have to.)

The `BEGIN` forces the `require` and `import` to happen at compile time. The `require` makes sure the module is loaded into memory if it hasn't been yet. The `import` is not a builtin--it's just an ordinary static method call into the `Module` package to tell the module to import the list of features back into the current package. The module can implement its `import` method any way it likes, though most modules just choose to derive their `import` method via inheritance from the `Exporter` class that is defined in the `Exporter` module. See *Exporter*. If no `import` method can be found then the call is skipped.

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).

There's a corresponding `no` command that unimports meanings imported by `use`, i.e., it calls `unimport Module LIST` instead of `import`.

```
    no integer;
    no strict 'refs';
    no warnings;
```

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 NUMERICAL 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 5.7.2, if the first two elements of the list are `undef`, then the utime(2) function in the C library will be 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 even work on other users' files where you have write permission:

```
    utime undef, undef, @ARGV;
```

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.

Note that only passing one of the first two elements as `undef` will be equivalent of passing it as 0 and will not have the same effect as described when they are both `undef`. This case will also trigger an uninitialized warning.

values HASH

Returns a list consisting of all the values of the named hash. (In a scalar context, returns the number of values.)

The values are returned in an apparently random order. The actual random order is subject to change in future versions of perl, but it is guaranteed to be the same order as either the `keys` or `each` function would produce on the same (unmodified) hash. Since Perl 5.8.1 the ordering is different even between different runs of Perl for security reasons (see *"Algorithmic Complexity Attacks" in perlsec*).

As a side effect, calling values() resets the HASH's internal iterator, see *each*. (In particular, calling values() in void context resets the iterator with no other overhead.)

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
```

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).

The string should not contain any character with the value > 255 (which can only happen if you're using UTF-8 encoding). If it does, it will be treated as something that is not UTF-8 encoded. When the `vec` was assigned to, other parts of your program will also no longer consider the string to be UTF-8 encoded. In other words, if you do have such characters in your string, vec() will operate on the actual byte string, and not the conceptual character string.

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 is run, the above example should print the following table:

```
                                       0         1         2
          3
                       unpack("V",$_)
01234567890123456789012345678901

-----------------------------------------------------------------
--
    vec($_, 0, 1) = 1   ==          1
10000000000000000000000000000000
    vec($_, 1, 1) = 1   ==          2
01000000000000000000000000000000
    vec($_, 2, 1) = 1   ==          4
00100000000000000000000000000000
    vec($_, 3, 1) = 1   ==          8
00010000000000000000000000000000
    vec($_, 4, 1) = 1   ==         16
00001000000000000000000000000000
    vec($_, 5, 1) = 1   ==         32
00000100000000000000000000000000
    vec($_, 6, 1) = 1   ==         64
00000010000000000000000000000000
    vec($_, 7, 1) = 1   ==        128
00000001000000000000000000000000
    vec($_, 8, 1) = 1   ==        256
00000000100000000000000000000000
    vec($_, 9, 1) = 1   ==        512
00000000010000000000000000000000
    vec($_,10, 1) = 1   ==       1024
```

```
00000000001000000000000000000000    vec($_,11, 1) = 1   == 
       2048 00000000000100000000000000000000
    vec($_,12, 1) = 1   ==       4096
00000000000010000000000000000000
    vec($_,13, 1) = 1   ==       8192
00000000000001000000000000000000
    vec($_,14, 1) = 1   ==      16384
00000000000000100000000000000000
    vec($_,15, 1) = 1   ==      32768
00000000000000010000000000000000
    vec($_,16, 1) = 1   ==      65536
00000000000000001000000000000000
    vec($_,17, 1) = 1   ==     131072
00000000000000000100000000000000
    vec($_,18, 1) = 1   ==     262144
00000000000000000010000000000000
    vec($_,19, 1) = 1   ==     524288
00000000000000000001000000000000
    vec($_,20, 1) = 1   ==    1048576
00000000000000000000100000000000
    vec($_,21, 1) = 1   ==    2097152
00000000000000000000010000000000
    vec($_,22, 1) = 1   ==    4194304
00000000000000000000001000000000
    vec($_,23, 1) = 1   ==    8388608
00000000000000000000000100000000
    vec($_,24, 1) = 1   ==   16777216
00000000000000000000000010000000
    vec($_,25, 1) = 1   ==   33554432
00000000000000000000000001000000
    vec($_,26, 1) = 1   ==   67108864
00000000000000000000000000100000
    vec($_,27, 1) = 1   ==  134217728
00000000000000000000000000010000
    vec($_,28, 1) = 1   ==  268435456
00000000000000000000000000001000
    vec($_,29, 1) = 1   ==  536870912
00000000000000000000000000000100
    vec($_,30, 1) = 1   == 1073741824
00000000000000000000000000000010
    vec($_,31, 1) = 1   == 2147483648
00000000000000000000000000000001
    vec($_, 0, 2) = 1   ==          1
10000000000000000000000000000000
    vec($_, 1, 2) = 1   ==          4
00100000000000000000000000000000
    vec($_, 2, 2) = 1   ==         16
00001000000000000000000000000000
    vec($_, 3, 2) = 1   ==         64
00000010000000000000000000000000
    vec($_, 4, 2) = 1   ==        256
00000000100000000000000000000000
    vec($_, 5, 2) = 1   ==       1024
00000000001000000000000000000000
    vec($_, 6, 2) = 1   ==       4096
00000000000010000000000000000000
```

```
    vec($_, 7, 2) = 1   ==      16384
00000000000000100000000000000000
    vec($_, 8, 2) = 1   ==      65536
00000000000000001000000000000000
    vec($_, 9, 2) = 1   ==     262144
00000000000000000010000000000000
    vec($_,10, 2) = 1   ==    1048576
00000000000000000000100000000000
    vec($_,11, 2) = 1   ==    4194304
00000000000000000000001000000000
    vec($_,12, 2) = 1   ==   16777216
00000000000000000000000010000000
    vec($_,13, 2) = 1   ==   67108864
00000000000000000000000000100000
    vec($_,14, 2) = 1   ==  268435456
00000000000000000000000000001000
    vec($_,15, 2) = 1   == 1073741824
00000000000000000000000000000010
    vec($_, 0, 2) = 2   ==          2
01000000000000000000000000000000
    vec($_, 1, 2) = 2   ==          8
00010000000000000000000000000000
    vec($_, 2, 2) = 2   ==         32
00000100000000000000000000000000
    vec($_, 3, 2) = 2   ==        128
00000001000000000000000000000000
    vec($_, 4, 2) = 2   ==        512
00000000010000000000000000000000
    vec($_, 5, 2) = 2   ==       2048
00000000000100000000000000000000
    vec($_, 6, 2) = 2   ==       8192
00000000000001000000000000000000
    vec($_, 7, 2) = 2   ==      32768
00000000000000010000000000000000
    vec($_, 8, 2) = 2   ==     131072
00000000000000000100000000000000
    vec($_, 9, 2) = 2   ==     524288
00000000000000000001000000000000
    vec($_,10, 2) = 2   ==    2097152
00000000000000000000010000000000
    vec($_,11, 2) = 2   ==    8388608
00000000000000000000000100000000
    vec($_,12, 2) = 2   ==   33554432
00000000000000000000000001000000
    vec($_,13, 2) = 2   ==  134217728
00000000000000000000000000010000
    vec($_,14, 2) = 2   ==  536870912
00000000000000000000000000000100
    vec($_,15, 2) = 2   == 2147483648
00000000000000000000000000000001
    vec($_, 0, 4) = 1   ==          1
10000000000000000000000000000000
    vec($_, 1, 4) = 1   ==         16
00001000000000000000000000000000
    vec($_, 2, 4) = 1   ==        256
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
00000000010000000000000000000000
    vec($_, 3, 4) = 2   ==       8192
00000000000001000000000000000000
    vec($_, 4, 4) = 2   ==     131072
00000000000000000100000000000000
    vec($_, 5, 4) = 2   ==    2097152
00000000000000000000010000000000
    vec($_, 6, 4) = 2   ==   33554432
00000000000000000000000001000000
    vec($_, 7, 4) = 2   ==  536870912
00000000000000000000000000000100
    vec($_, 0, 4) = 4   ==          4
00100000000000000000000000000000
    vec($_, 1, 4) = 4   ==         64
00000010000000000000000000000000
    vec($_, 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
00000000000100000000000000000000
    vec($_, 3, 4) = 8   ==      32768
00000000000000010000000000000000
    vec($_, 4, 4) = 8   ==     524288
00000000000000000001000000000000
    vec($_, 5, 4) = 8   ==    8388608
00000000000000000000000100000000
    vec($_, 6, 4) = 8   ==  134217728
00000000000000000000000000010000
```

```
    vec($_, 7, 4) = 8   == 2147483648
00000000000000000000000000000001
    vec($_, 0, 8) = 1   ==          1
10000000000000000000000000000000
    vec($_, 1, 8) = 1   ==        256
00000000100000000000000000000000
    vec($_, 2, 8) = 1   ==      65536
00000000000000001000000000000000
    vec($_, 3, 8) = 1   ==   16777216
00000000000000000000000010000000
    vec($_, 0, 8) = 2   ==          2
01000000000000000000000000000000
    vec($_, 1, 8) = 2   ==        512
00000000010000000000000000000000
    vec($_, 2, 8) = 2   ==     131072
00000000000000000100000000000000
    vec($_, 3, 8) = 2   ==   33554432
00000000000000000000000001000000
    vec($_, 0, 8) = 4   ==          4
00100000000000000000000000000000
    vec($_, 1, 8) = 4   ==       1024
00000000001000000000000000000000
    vec($_, 2, 8) = 4   ==     262144
00000000000000000010000000000000
    vec($_, 3, 8) = 4   ==   67108864
00000000000000000000000000100000
    vec($_, 0, 8) = 8   ==          8
00010000000000000000000000000000
    vec($_, 1, 8) = 8   ==       2048
00000000000100000000000000000000
    vec($_, 2, 8) = 8   ==     524288
00000000000000000001000000000000
    vec($_, 3, 8) = 8   ==  134217728
00000000000000000000000000010000
    vec($_, 0, 8) = 16  ==         16
00001000000000000000000000000000
    vec($_, 1, 8) = 16  ==       4096
00000000000010000000000000000000
    vec($_, 2, 8) = 16  ==    1048576
00000000000000000000100000000000
    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 the wait(2) system call 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 $?. Note that a return value of `-1` could mean that child processes are being automatically reaped, as described in *perlipc*.

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 $?. If you say

```
    use POSIX ":sys_wait_h";
    #...
    do {
 $kid = waitpid(-1, WNOHANG);
    } until $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) system calls. However, waiting for a particular pid with FLAGS of `0` is implemented everywhere. (Perl emulates the system call by remembering the status values of processes that have exited but have not been harvested by the Perl script yet.)

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.

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`, `CHECK`, `INIT` or `END` block, or in a `DESTROY` method.

This function should have been named wantlist() instead.

warn LIST

Produces a message on STDERR just like `die`, but doesn't exit or throw an exception.

If LIST is empty and $@ already contains a value (typically from a previous eval) that value is used after appending `"\t...caught"` to $@. This is useful for staying almost, but not entirely similar to `die`.

If $@ is empty then the string `"Warning: Something's wrong"` is used.

No message is printed if there is a `$SIG{__WARN__}` handler installed. It is the handler's responsibility to deal with the message as it sees fit (like, for instance, converting it into a `die`). Most handlers must therefore make arrangements to actually display the warnings that they are not prepared to deal with, by calling `warn` again in the handler. Note that this is quite safe and will not produce an endless loop, since `__WARN__` hooks are not called from inside one.

You will find this behavior is slightly different from that of `$SIG{__DIE__}` handlers (which don't suppress the error text, but can instead call `die` again to change it).

Using a `__WARN__` handler provides a powerful way to silence all warnings (even the so-called mandatory ones). An example:

```
# wipe out *all* compile-time warnings
BEGIN { $SIG{'__WARN__'} = sub { warn $_[0] if $DOWARN } }
my $foo = 10;
my $foo = 20;          # no warning about duplicate my $foo,
                       # but hey, you asked for it!
# no compile-time or run-time warnings before here
$DOWARN = 1;

# run-time warnings enabled after here
warn "\$foo is alive and $foo!";     # does show up
```

See *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, a special top-of-page format is used to format the new page header, and then the record is written. By default the top-of-page format is the name of the filehandle with "_TOP" appended, but it may be dynamically set to the format of your choice by assigning the name to the `$^` variable while the filehandle is selected. The number of lines remaining on the current page is in variable `$-`, which can be set to `0` to force a new page.

If FILEHANDLE is unspecified, output goes to the current default output channel, which starts out as STDOUT but may be changed by the `select` operator. If the FILEHANDLE is an EXPR, then the expression is evaluated and the resulting string is used to look up the name of the FILEHANDLE at run time. For more on formats, see *perlform*.

Note that write is *not* the opposite of `read`. Unfortunately.

y///

The transliteration operator. Same as `tr///`. See *perlop*.