

NAME

perlvar – Perl predefined variables

DESCRIPTION

Predefined Names

The following names have special meaning to Perl. Most punctuation names have reasonable mnemonics, or analogues in one of the shells. Nevertheless, if you wish to use long variable names, you just need to say

```
use English;
```

at the top of your program. This will alias all the short names to the long names in the current package. Some even have medium names, generally borrowed from **awk**.

To go a step further, those variables that depend on the currently selected filehandle may instead (and preferably) be set by calling an object method on the FileHandle object. (Summary lines below for this contain the word HANDLE.) First you must say

```
use FileHandle;
```

after which you may use either

```
method HANDLE EXPR
```

or more safely,

```
HANDLE->method( EXPR )
```

Each of the methods returns the old value of the FileHandle attribute. The methods each take an optional EXPR, which if supplied specifies the new value for the FileHandle attribute in question. If not supplied, most of the methods do nothing to the current value, except for *autoflush()*, which will assume a 1 for you, just to be different.

A few of these variables are considered “read-only”. This means that if you try to assign to this variable, either directly or indirectly through a reference, you’ll raise a run-time exception.

The following list is ordered by scalar variables first, then the arrays, then the hashes (except $\M was added in the wrong place). This is somewhat obscured by the fact that %ENV and %SIG are listed as \$ENV{expr} and \$SIG{expr}.

\$ARG

\$_ The default input and pattern-searching space. The following pairs are equivalent:

```
while (<>) {...}      # equivalent in only while!
while (defined($_ = <>)) {...}

/^Subject:/
$_ =~ /^Subject:/
```

```
tr/a-z/A-Z/
$_ =~ tr/a-z/A-Z/
```

```
chop
chop($_)
```

Here are the places where Perl will assume `$_` even if you don't use it:

- Various unary functions, including functions like *ord()* and *int()*, as well as the all file tests (*-f*, *-d*) except for *-t*, which defaults to STDIN.
- Various list functions like *print()* and *unlink()*.
- The pattern matching operations *m//*, *s//*, and *tr//* when used without an `=~` operator.
- The default iterator variable in a *foreach* loop if no other variable is supplied.
- The implicit iterator variable in the *grep()* and *map()* functions.
- The default place to put an input record when a `<FH>` operation's result is tested by itself as the sole criterion of a *while* test. Note that outside of a *while* test, this will not happen.

(Mnemonic: underline is understood in certain operations.)

`$<digits>`

Contains the subpattern from the corresponding set of parentheses in the last pattern matched, not counting patterns matched in nested blocks that have been exited already. (Mnemonic: like `\digits`.) These variables are all read-only.

`$MATCH`

`$&` The string matched by the last successful pattern match (not counting any matches hidden within a BLOCK or *eval()* enclosed by the current BLOCK). (Mnemonic: like `&` in some editors.) This variable is read-only.

`$PREMATCH`

`$'` The string preceding whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or *eval* enclosed by the current BLOCK). (Mnemonic: `'` often precedes a quoted string.) This variable is read-only.

`$POSTMATCH`

`$'` The string following whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or *eval()* enclosed by the current BLOCK). (Mnemonic: `'` often follows a quoted string.)

Example:

```
$_ = 'abcdefghi';
/def/;
print "$`:$&:$'\n";           # prints abc:def:ghi
```

This variable is read-only.

\$LAST_PAREN_MATCH

\$+ The last bracket matched by the last search pattern. This is useful if you don't know which of a set of alternative patterns matched. For example:

```
/Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

(Mnemonic: be positive and forward looking.) This variable is read-only.

\$MULTILINE_MATCHING

\$* Set to 1 to do multi-line matching within a string, 0 to tell Perl that it can assume that strings contain a single line, for the purpose of optimizing pattern matches. Pattern matches on strings containing multiple newlines can produce confusing results when “\$*” is 0. Default is 0. (Mnemonic: * matches multiple things.) Note that this variable influences the interpretation of only “^” and “\$”. A literal newline can be searched for even when \$* == 0.

Use of “\$*” is deprecated in modern Perls, supplanted by the /s and /m modifiers on pattern matching.

input_line_number HANDLE EXPR**\$INPUT_LINE_NUMBER****\$NR**

\$. The current input line number for the last file handle from which you read (or performed a `seek` or `tell` on). An explicit close on a filehandle resets the line number. Because “<>” never does an explicit close, line numbers increase across ARGV files (but see examples under *eof()*). Localizing \$. has the effect of also localizing Perl's notion of “the last read filehandle”. (Mnemonic: many programs use “.” to mean the current line number.)

input_record_separator HANDLE EXPR**\$INPUT_RECORD_SEPARATOR****\$RS**

\$/ The input record separator, newline by default. Works like **awk**'s **RS** variable, including treating empty lines as delimiters if set to the null string. (Note: An empty line cannot contain any spaces or tabs.) You may set it to a multi-character string to match a multi-character delimiter, or to `undef` to read to end of file. Note that setting it to “\n\n” means something slightly different than setting it to “ ”, if the file contains consecutive empty lines. Setting it to “ ” will treat two or more consecutive empty lines as a single empty line. Setting it to “\n\n” will blindly assume that the next input character belongs to the next paragraph, even if it's a newline. (Mnemonic: / is used to delimit line boundaries when quoting poetry.)

```
undef $/;
$_ = <FH>;           # whole file now here
s/\n[ \t]+/ /g;
```

Remember: the value of **\$/** is a string, not a regexp. AWK has to be better for something :-)

Setting `$/` to a reference to an integer, scalar containing an integer, or scalar that's convertible to an integer will attempt to read records instead of lines, with the maximum record size being the referenced integer. So this:

```
$/ = \32768; # or \"32768", or \${var_containing_32768}
open(FILE, $myfile);
$_ = <FILE>;
```

will read a record of no more than 32768 bytes from `FILE`. If you're not reading from a record-oriented file (or your OS doesn't have record-oriented files), then you'll likely get a full chunk of data with every read. If a record is larger than the record size you've set, you'll get the record back in pieces.

On VMS, record reads are done with the equivalent of `sysread`, so it's best not to mix record and non-record reads on the same file. (This is likely not a problem, as any file you'd want to read in record mode is probably usable in line mode) Non-VMS systems perform normal I/O, so it's safe to mix record and non-record reads of a file.

autoflush HANDLE_EXPR

\$OUTPUT_AUTOFLUSH

\$| If set to nonzero, forces a flush right away and after every write or print on the currently selected output channel. Default is 0 (regardless of whether the channel is actually buffered by the system or not; `$|` tells you only whether you've asked Perl explicitly to flush after each write). Note that `STDOUT` will typically be line buffered if output is to the terminal and block buffered otherwise. Setting this variable is useful primarily when you are outputting to a pipe, such as when you are running a Perl script under `rsh` and want to see the output as it's happening. This has no effect on input buffering. (Mnemonic: when you want your pipes to be piping hot.)

output_field_separator HANDLE_EXPR

\$OUTPUT_FIELD_SEPARATOR

\$OFS

\$, The output field separator for the print operator. Ordinarily the print operator simply prints out the comma-separated fields you specify. To get behavior more like **awk**, set this variable as you would set **awk**'s `OFS` variable to specify what is printed between fields. (Mnemonic: what is printed when there is a , in your print statement.)

output_record_separator HANDLE_EXPR

\$OUTPUT_RECORD_SEPARATOR

\$ORS

\$\$ The output record separator for the print operator. Ordinarily the print operator simply prints out the comma-separated fields you specify, with no trailing newline or record separator assumed. To get behavior more like **awk**, set this variable as you would set **awk**'s `ORS` variable to specify what is printed at the end of the print. (Mnemonic: you set "\$\`" instead of adding `\n` at the end of the print. Also, it's just like `$/`, but it's what you get "back" from Perl.)

\$LIST_SEPARATOR

\$ This is like “\$,” except that it applies to array values interpolated into a double-quoted string (or similar interpreted string). Default is a space. (Mnemonic: obvious, I think.)

\$SUBSCRIPT_SEPARATOR**\$SUBSEP**

\$; The subscript separator for multidimensional array emulation. If you refer to a hash element as

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

it really means

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

But don't put

```
@foo{$a,$b,$c}      # a slice--note the @
```

which means

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

Default is “\034”, the same as SUBSEP in **awk**. Note that if your keys contain binary data there might not be any safe value for “\$;”. (Mnemonic: comma (the syntactic subscript separator) is a semi-semicolon. Yeah, I know, it's pretty lame, but “\$,” is already taken for something more important.)

Consider using “real” multidimensional arrays.

\$OFMT

\$# The output format for printed numbers. This variable is a half-hearted attempt to emulate **awk**'s OFMT variable. There are times, however, when **awk** and Perl have differing notions of what is in fact numeric. The initial value is *%ng*, where *n* is the value of the macro DBL_DIG from your system's *float.h*. This is different from **awk**'s default OFMT setting of *%.6g*, so you need to set “\$#” explicitly to get **awk**'s value. (Mnemonic: # is the number sign.)

Use of “\$#” is deprecated.

format_page_number HANDLE EXPR**\$FORMAT_PAGE_NUMBER**

% The current page number of the currently selected output channel. (Mnemonic: % is page number in **nroff**.)

format_lines_per_page HANDLE EXPR**\$FORMAT_LINES_PER_PAGE**

\$= The current page length (printable lines) of the currently selected output channel. Default is 60. (Mnemonic: = has horizontal lines.)

format_lines_left HANDLE EXPR

\$FORMAT_LINES_LEFT

\$- The number of lines left on the page of the currently selected output channel. (Mnemonic: lines_on_page – lines_printed.)

format_name HANDLE EXPR

\$FORMAT_NAME

\$~ The name of the current report format for the currently selected output channel. Default is name of the filehandle. (Mnemonic: brother to “\$^”.)

format_top_name HANDLE EXPR

\$FORMAT_TOP_NAME

\$^ The name of the current top-of-page format for the currently selected output channel. Default is name of the filehandle with `_TOP` appended. (Mnemonic: points to top of page.)

format_line_break_characters HANDLE EXPR

\$FORMAT_LINE_BREAK_CHARACTERS

\$: The current set of characters after which a string may be broken to fill continuation fields (starting with `^`) in a format. Default is “\n-”, to break on whitespace or hyphens. (Mnemonic: a “colon” in poetry is a part of a line.)

format_formfeed HANDLE EXPR

\$FORMAT_FORMFEED

\$\$L What formats output to perform a form feed. Default is `\f`.

\$ACCUMULATOR

\$\$A The current value of the `write()` accumulator for `format()` lines. A format contains `formline()` commands that put their result into `$$A`. After calling its format, `write()` prints out the contents of `$$A` and empties. So you never actually see the contents of `$$A` unless you call `formline()` yourself and then look at it. See the `perform` manpage and the `formline()` entry in the `perlfunc` manpage.

\$CHILD_ERROR

\$\$? The status returned by the last pipe close, backtick (`` ``) command, or `system()` operator. Note that this is the status word returned by the `wait()` system call (or else is made up to look like it). Thus, the exit value of the subprocess is actually (`$$? >> 8`), and `$$? & 127` gives which signal, if any, the process died from, and `$$? & 128` reports whether there was a core dump. (Mnemonic: similar to **sh** and **ksh**.)

Additionally, if the `h_errno` variable is supported in C, its value is returned via `$$?` if any of the `gethost*()` functions fail.

Note that if you have installed a signal handler for `SIGCHLD`, the value of `$$?` will usually be wrong outside that handler.

Inside an `END` subroutine `$?` contains the value that is going to be given to `exit()`. You can modify `$?` in an `END` subroutine to change the exit status of the script.

Under VMS, the pragma `use vmsish 'status'` makes `$?` reflect the actual VMS exit status, instead of the default emulation of POSIX status.

Also see the section on *Error Indicators*.

\$OS_ERROR

\$ERRNO

\$! If used in a numeric context, yields the current value of `errno`, with all the usual caveats. (This means that you shouldn't depend on the value of `$!` to be anything in particular unless you've gotten a specific error return indicating a system error.) If used in a string context, yields the corresponding system error string. You can assign to `$!` to set `errno` if, for instance, you want "`$!`" to return the string for error `n`, or you want to set the exit value for the `die()` operator. (Mnemonic: What just went bang?)

Also see the section on *Error Indicators*.

\$EXTENDED_OS_ERROR

\$\$E Error information specific to the current operating system. At the moment, this differs from `$!` under only VMS, OS/2, and Win32 (and for MacPerl). On all other platforms, `$$E` is always just the same as `$!`.

Under VMS, `$$E` provides the VMS status value from the last system error. This is more specific information about the last system error than that provided by `$!`. This is particularly important when `$!` is set to **EVMSEERR**.

Under OS/2, `$$E` is set to the error code of the last call to OS/2 API either via CRT, or directly from perl.

Under Win32, `$$E` always returns the last error information reported by the Win32 call `GetLastError()` which describes the last error from within the Win32 API. Most Win32-specific code will report errors via `$$E`. ANSI C and UNIX-like calls set `errno` and so most portable Perl code will report errors via `$!`.

Caveats mentioned in the description of `$!` generally apply to `$$E`, also. (Mnemonic: Extra error explanation.)

Also see the section on *Error Indicators*.

\$EVAL_ERROR

\$\$@ The Perl syntax error message from the last `eval()` command. If null, the last `eval()` parsed and executed correctly (although the operations you invoked may have failed in the normal fashion). (Mnemonic: Where was the syntax error "at"?)

Note that warning messages are not collected in this variable. You can, however, set up a routine to process warnings by setting `$$SIG{__WARN__}` as described below.

Also see the section on *Error Indicators*.

\$PROCESS_ID**\$PID**

\$\$ The process number of the Perl running this script. (Mnemonic: same as shells.)

\$REAL_USER_ID**\$UID**

\$< The real uid of this process. (Mnemonic: it's the uid you came *FROM*, if you're running *setuid*.)

\$EFFECTIVE_USER_ID**\$EUID**

\$> The effective uid of this process. Example:

```
$< = $>;           # set real to effective uid
($<, $>) = ($>, $<); # swap real and effective uid
```

(Mnemonic: it's the uid you went *TO*, if you're running *setuid*.) Note: “\$<” and “\$>” can be swapped only on machines supporting *setreuid*().

\$REAL_GROUP_ID**\$GID**

\$(The real gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by *getgid*(), and the subsequent ones by *getgroups*(), one of which may be the same as the first number.

However, a value assigned to “\$(” must be a single number used to set the real gid. So the value given by “\$(” should *not* be assigned back to “\$(” without being forced numeric, such as by adding zero.

(Mnemonic: parentheses are used to *GROUP* things. The real gid is the group you *LEFT*, if you're running *setgid*.)

\$EFFECTIVE_GROUP_ID**\$EGID**

\$) The effective gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by *getegid*(), and the subsequent ones by *getgroups*(), one of which may be the same as the first number.

Similarly, a value assigned to “\$)” must also be a space-separated list of numbers. The first number is used to set the effective gid, and the rest (if any) are passed to *setgroups*(). To get the effect of an empty list for *setgroups*(), just repeat the new effective gid; that is, to force an effective gid of 5 and an effectively empty *setgroups*() list, say `$) = "5 5" .`

(Mnemonic: parentheses are used to *GROUP* things. The effective gid is the group that's *RIGHT* for you, if you're running *setgid*.)

Note: “\$<”, “\$>”, “\$(” and “\$)” can be set only on machines that support the corresponding *set[re][ug]id()* routine. “\$(” and “\$)” can be swapped only on machines supporting *setregid()*.

\$PROGRAM_NAME

\$0 Contains the name of the file containing the Perl script being executed. On some operating systems assigning to “\$0” modifies the argument area that the *ps* (1) program sees. This is more useful as a way of indicating the current program state than it is for hiding the program you’re running. (Mnemonic: same as **sh** and **ksh**.)

[\$ The index of the first element in an array, and of the first character in a substring. Default is 0, but you could set it to 1 to make Perl behave more like **awk** (or Fortran) when subscripting and when evaluating the *index()* and *substr()* functions. (Mnemonic: [begins subscripts.)

As of Perl 5, assignment to “\$[” is treated as a compiler directive, and cannot influence the behavior of any other file. Its use is discouraged.

\$PERL_VERSION

[\$] The version + patchlevel / 1000 of the Perl interpreter. This variable can be used to determine whether the Perl interpreter executing a script is in the right range of versions. (Mnemonic: Is this version of perl in the right bracket?) Example:

```
warn "No checksumming!\n" if $] < 3.019;
```

See also the documentation of `use VERSION` and `require VERSION` for a convenient way to fail if the Perl interpreter is too old.

\$DEBUGGING

[\$^D The current value of the debugging flags. (Mnemonic: value of **-D** switch.)

\$SYSTEM_FD_MAX

[\$^F The maximum system file descriptor, ordinarily 2. System file descriptors are passed to *exec()*ed processes, while higher file descriptors are not. Also, during an *open()*, system file descriptors are preserved even if the *open()* fails. (Ordinary file descriptors are closed before the *open()* is attempted.) Note that the close-on-exec status of a file descriptor will be decided according to the value of **[\$^F** at the time of the open, not the time of the exec.

[\$^H The current set of syntax checks enabled by `use strict` and other block scoped compiler hints. See the documentation of `strict` for more details.

\$INPLACE_EDIT

[\$^I The current value of the inplace-edit extension. Use `undef` to disable inplace editing. (Mnemonic: value of **-i** switch.)

[\$^M By default, running out of memory it is not trappable. However, if compiled for this, Perl may use the contents of **[\$^M** as an emergency pool after *die()*ing with this message. Suppose that your Perl were compiled with `-DPERL_EMERGENCY_SBRK` and used Perl’s `malloc`. Then

```
^M = 'a' x (1<<16);
```

would allocate a 64K buffer for use when in emergency. See the *INSTALL* file for information on how to enable this option. As a disincentive to casual use of this advanced feature, there is no the *English* manpage long name for this variable.

\$OSNAME

^O The name of the operating system under which this copy of Perl was built, as determined during the configuration process. The value is identical to `$Config{ 'osname' }`.

\$PERLDB

^P The internal variable for debugging support. Different bits mean the following (subject to change):

0x01 Debug subroutine enter/exit.

0x02 Line-by-line debugging.

0x04 Switch off optimizations.

0x08 Preserve more data for future interactive inspections.

0x10 Keep info about source lines on which a subroutine is defined.

0x20 Start with single-step on.

Note that some bits may be relevant at compile-time only, some at run-time only. This is a new mechanism and the details may change.

^R The result of evaluation of the last successful the section on (? { code }) in the *perlre* manpage regular expression assertion. (Excluding those used as switches.) May be written to.

^S Current state of the interpreter. Undefined if parsing of the current module/eval is not finished (may happen in `$SIG{__DIE__}` and `$SIG{__WARN__}` handlers). True if inside an eval, otherwise false.

\$BASETIME

^T The time at which the script began running, in seconds since the epoch (beginning of 1970). The values returned by the `-M`, `-A`, and `-C` filetests are based on this value.

\$WARNING

^W The current value of the warning switch, either TRUE or FALSE. (Mnemonic: related to the `-w` switch.)

\$EXECUTABLE_NAME

^X The name that the Perl binary itself was executed as, from C's `argv[0]`.

\$ARGV contains the name of the current file when reading from `<>`.

@ARGV The array `@ARGV` contains the command line arguments intended for the script. Note that `$#ARGV` is the generally number of arguments minus one, because `$ARGV[0]` is the first argument, *NOT* the command name. See “\$0” for the command name.

@INC The array @INC contains the list of places to look for Perl scripts to be evaluated by the `do` *EXPR*, `require`, or `use` constructs. It initially consists of the arguments to any **-I** command line switches, followed by the default Perl library, probably `/usr/local/lib/perl`, followed by `“.”`, to represent the current directory. If you need to modify this at runtime, you should use the `use lib` pragma to get the machine-dependent library properly loaded also:

```
use lib '/mypath/libdir/';
use SomeMod;
```

@_ Within a subroutine the array @_ contains the parameters passed to that subroutine. See the *perlsub* manpage.

%INC The hash %INC contains entries for each filename that has been included via `do` or `require`. The key is the filename you specified, and the value is the location of the file actually found. The `require` command uses this array to determine whether a given file has already been included.

%ENV \$ENV{expr}

The hash %ENV contains your current environment. Setting a value in ENV changes the environment for child processes.

%SIG \$SIG{expr}

The hash %SIG is used to set signal handlers for various signals. Example:

```
sub handler {          # 1st argument is signal name
    my($sig) = @_;
    print "Caught a SIG$sig--shutting down\n";
    close(LOG);
    exit(0);
}

$SIG{'INT'} = \&handler;
$SIG{'QUIT'} = \&handler;
...
$SIG{'INT'} = 'DEFAULT';    # restore default action
$SIG{'QUIT'} = 'IGNORE';   # ignore SIGQUIT
```

The %SIG array contains values for only the signals actually set within the Perl script. Here are some other examples:

```
$SIG{"PIPE"} = Plumber;    # SCARY!!
$SIG{"PIPE"} = "Plumber";  # assumes main::Plumber (not record)
$SIG{"PIPE"} = \&Plumber;  # just fine; assume current Plumber
$SIG{"PIPE"} = Plumber();  # oops, what did Plumber() return
```

The one marked scary is problematic because it's a bareword, which means sometimes it's a string representing the function, and sometimes it's going to call the subroutine call right then and there! Best to be sure and quote it or take a reference to it. `*Plumber` works too. See the *perlsub* manpage.

If your system has the *sigaction()* function then signal handlers are installed using it. This means you get reliable signal handling. If your system has the SA_RESTART flag it is used when signals handlers are installed. This means

that system calls for which it is supported continue rather than returning when a signal arrives. If you want your system calls to be interrupted by signal delivery then do something like this:

```
use POSIX ':signal_h';

my $alarm = 0;
sigaction SIGALRM, new POSIX::SigAction sub { $alarm = 1 }
    or die "Error setting SIGALRM handler: $!\n";
```

See the *POSIX* manpage.

Certain internal hooks can be also set using the %SIG hash. The routine indicated by \$SIG{__WARN__} is called when a warning message is about to be printed. The warning message is passed as the first argument. The presence of a __WARN__ hook causes the ordinary printing of warnings to STDERR to be suppressed. You can use this to save warnings in a variable, or turn warnings into fatal errors, like this:

```
local $SIG{__WARN__} = sub { die $_[0] };
eval $proggie;
```

The routine indicated by \$SIG{__DIE__} is called when a fatal exception is about to be thrown. The error message is passed as the first argument. When a __DIE__ hook routine returns, the exception processing continues as it would have in the absence of the hook, unless the hook routine itself exits via a goto, a loop exit, or a die(). The __DIE__ handler is explicitly disabled during the call, so that you can die from a __DIE__ handler. Similarly for __WARN__.

Note that the \$SIG{__DIE__} hook is called even inside eval()ed blocks/strings. See the die entry in the *perlfunc* manpage and the section on \$^S in the *perlvar* manpage for how to circumvent this.

Note that __DIE__/__WARN__ handlers are very special in one respect: they may be called to report (probable) errors found by the parser. In such a case the parser may be in inconsistent state, so any attempt to evaluate Perl code from such a handler will probably result in a segfault. This means that calls which result/may-result in parsing Perl should be used with extreme caution, like this:

```
require Carp if defined $^S;
Carp::confess("Something wrong") if defined &Carp::confess;
die "Something wrong, but could not load Carp to give backtrace";
    To see backtrace try starting Perl with -MCarp switch";
```

Here the first line will load Carp *unless* it is the parser who called the handler. The second line will print backtrace and die if Carp was available. The third line will be executed only if Carp was not available.

See the die entry in the *perlfunc* manpage, the warn entry in the *perlfunc* manpage and the eval entry in the *perlfunc* manpage for additional info.

Error Indicators

The variables the section on `$@`, the section on `$!`, the section on `^E`, and the section on `$?` contain information about different types of error conditions that may appear during execution of Perl script. The variables are shown ordered by the “distance” between the subsystem which reported the error and the Perl process, and correspond to errors detected by the Perl interpreter, C library, operating system, or an external program, respectively.

To illustrate the differences between these variables, consider the following Perl expression:

```
eval '
    open PIPE, "/cdrom/install |";
    @res = <PIPE>;
    close PIPE or die "bad pipe: $?, $!";
';
```

After execution of this statement all 4 variables may have been set.

`$@` is set if the string to be `eval`-ed did not compile (this may happen if `open` or `close` were imported with bad prototypes), or if Perl code executed during evaluation `die()`d (either implicitly, say, if `open` was imported from module the *Fatal* manpage, or the `die` after `close` was triggered). In these cases the value of `$@` is the compile error, or *Fatal* error (which will interpolate `$!`), or the argument to `die` (which will interpolate `$!` and `$?!`).

When the above expression is executed, `open()`, `<PIPE>`, and `close` are translated to C run-time library calls. `$!` is set if one of these calls fails. The value is a symbolic indicator chosen by the C run-time library, say `No such file or directory`.

On some systems the above C library calls are further translated to calls to the kernel. The kernel may have set more verbose error indicator than one of the handful of standard C errors. In such cases `^E` contains this verbose error indicator, which may be, say, `CDROM tray not closed`. On systems where C library calls are identical to system calls `^E` is a duplicate of `$!`.

Finally, `$?` may be set to non-0 value if the external program `/cdrom/install` fails. Upper bits of the particular value may reflect specific error conditions encountered by this program (this is program-dependent), lower-bits reflect mode of failure (segfault, completion, etc.). Note that in contrast to `$@`, `$!`, and `^E`, which are set only if error condition is detected, the variable `$?` is set on each `wait` or pipe `close`, overwriting the old value.

For more details, see the individual descriptions at the section on `$@`, the section on `$!`, the section on `^E`, and the section on `$?`.

