NAME

perlvar - Perl predefined variables

DESCRIPTION

The Syntax of Variable Names

Variable names in Perl can have several formats. Usually, they must begin with a letter or underscore, in which case they can be arbitrarily long (up to an internal limit of 251 characters) and may contain letters, digits, underscores, or the special sequence `::` or `'`. In this case, the part before the last `::` or `'` is taken to be a *package qualifier*; see *perlmod*. A Unicode letter that is not ASCII is not considered to be a letter unless `"use utf8"` is in effect, and somewhat more complicated rules apply; see *"Identifier parsing" in perldata* for details.

Perl variable names may also be a sequence of digits, a single punctuation character, or the two-character sequence: `^` (caret or CIRCUMFLEX ACCENT) followed by any one of the characters `[][A-Z^_?\]`. These names are all reserved for special uses by Perl; for example, the all-digits names are used to hold data captured by backreferences after a regular expression match.

Since Perl v5.6.0, Perl variable names may also be alphanumeric strings preceded by a caret. These must all be written in the form `${^Foo}`; the braces are not optional. `${^Foo}` denotes the scalar variable whose name is considered to be a control-F followed by two o's. These variables are reserved for future special uses by Perl, except for the ones that begin with `^_` (caret-underscore). No name that begins with `^_` will acquire a special meaning in any future version of Perl; such names may therefore be used safely in programs. `$^_` itself, however, *is* reserved.

Perl identifiers that begin with digits or punctuation characters are exempt from the effects of the `package` declaration and are always forced to be in package `main`; they are also exempt from `strict 'vars'` errors. A few other names are also exempt in these ways:

```
ENV      STDIN
INC      STDOUT
ARGV     STDERR
ARGVOUT
SIG
```

In particular, the special `${^_XYZ}` variables are always taken to be in package `main`, regardless of any `package` declarations presently in scope.

SPECIAL VARIABLES

The following names have special meaning to Perl. Most punctuation names have reasonable mnemonics, or analogs in the shells. Nevertheless, if you wish to use long variable names, you need only say:

```
use English;
```

at the top of your program. This aliases all the short names to the long names in the current package. Some even have medium names, generally borrowed from **awk**. For more info, please see *English*.

Before you continue, note the sort order for variables. In general, we first list the variables in case-insensitive, almost-lexigraphical order (ignoring the `{` or `^` preceding words, as in `${^UNICODE}` or `$^T`), although `$_` and `@_` move up to the top of the pile. For variables with the same identifier, we list it in order of scalar, array, hash, and bareword.

General Variables

$ARG

$_

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

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

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

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

chomp
chomp($_)
```

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

- The following functions use $_ as a default argument:

  abs, alarm, chomp, chop, chr, chroot, cos, defined, eval, evalbytes, exp, fc, glob, hex, int, lc, lcfirst, length, log, lstat, mkdir, oct, ord, pos, print, printf, quotemeta, readlink, readpipe, ref, require, reverse (in scalar context only), rmdir, say, sin, split (for its second argument), sqrt, stat, study, uc, ucfirst, unlink, unpack.

- All file tests (`-f`, `-d`) except for `-t`, which defaults to STDIN. See *"-X" in perlfunc*

- The pattern matching operations `m//`, `s///` and `tr///` (aka `y///`) when used without an =~ operator.

- The default iterator variable in a `foreach` loop if no other variable is supplied.

- The implicit iterator variable in the `grep()` and `map()` functions.

- The implicit variable of `given()`.

- The default place to put the next value or input record when a `<FH>`, `readline`, `readdir` or `each` operation's result is tested by itself as the sole criterion of a `while` test. Outside a `while` test, this will not happen.

$_ is by default a global variable. However, as of perl v5.10.0, you can use a lexical version of $_ by declaring it in a file or in a block with `my`. Moreover, declaring `our $_` restores the global $_ in the current scope. Though this seemed like a good idea at the time it was introduced, lexical $_ actually causes more problems than it solves. If you call a function that expects to be passed information via $_, it may or may not work, depending on how the function is written, there not being any easy way to solve this. Just avoid lexical $_, unless you are feeling particularly masochistic. For this reason lexical $_ is still experimental and will produce a warning unless warnings have been disabled. As with other experimental features, the behavior of lexical $_ is subject to change without notice, including change into a fatal error.

Mnemonic: underline is understood in certain operations.

@ARG

@_

Within a subroutine the array `@_` contains the parameters passed to that subroutine. Inside a subroutine, `@_` is the default array for the array operators `pop` and `shift`.

See *perlsub*.

$LIST_SEPARATOR

$"

When an array or an array slice is interpolated into a double-quoted string or a similar context such as `/.../`, its elements are separated by this value. Default is a space.

For example, this:

```
print "The array is: @array\n";
```

is equivalent to this:

```
print "The array is: " . join($", @array) . "\n";
```

Mnemonic: works in double-quoted context.

$PROCESS_ID

$PID

$$

The process number of the Perl running this script. Though you *can* set this variable, doing so is generally discouraged, although it can be invaluable for some testing purposes. It will be reset automatically across `fork()` calls.

Note for Linux and Debian GNU/kFreeBSD users: Before Perl v5.16.0 perl would emulate POSIX semantics on Linux systems using LinuxThreads, a partial implementation of POSIX Threads that has since been superseded by the Native POSIX Thread Library (NPTL).

LinuxThreads is now obsolete on Linux, and caching `getpid()` like this made embedding perl unnecessarily complex (since you'd have to manually update the value of $$), so now `$$` and `getppid()` will always return the same values as the underlying C library.

Debian GNU/kFreeBSD systems also used LinuxThreads up until and including the 6.0 release, but after that moved to FreeBSD thread semantics, which are POSIX-like.

To see if your system is affected by this discrepancy check if `getconf GNU_LIBPTHREAD_VERSION | grep -q NPTL` returns a false value. NTPL threads preserve the POSIX semantics.

Mnemonic: same as shells.

$PROGRAM_NAME

$0

Contains the name of the program being executed.

On some (but not all) operating systems assigning to `$0` modifies the argument area that the `ps` program sees. On some platforms you may have to use special `ps` options or a different `ps` to see the changes. Modifying the `$0` is more useful as a way of indicating the current program state than it is for hiding the program you're running.

Note that there are platform-specific limitations on the maximum length of `$0`. In the most extreme case it may be limited to the space occupied by the original `$0`.

In some platforms there may be arbitrary amount of padding, for example space characters, after the modified name as shown by `ps`. In some platforms this padding may extend all the way to the original length of the argument area, no matter what you do (this is the case for example with Linux 2.2).

Note for BSD users: setting `$0` does not completely remove "perl" from the ps(1) output. For example, setting `$0` to `"foobar"` may result in `"perl: foobar (perl)"` (whether both the `"perl: "` prefix and the " (perl)" suffix are shown depends on your exact BSD variant and version). This is an operating system feature, Perl cannot help it.

In multithreaded scripts Perl coordinates the threads so that any thread may modify its copy of the `$0` and the change becomes visible to ps(1) (assuming the operating system plays along). Note that the view of `$0` the other threads have will not change since they have their own copies of it.

If the program has been given to perl via the switches `-e` or `-E`, `$0` will contain the string `"-e"`.

On Linux as of perl v5.14.0 the legacy process name will be set with `prctl(2)`, in addition to altering the POSIX name via `argv[0]` as perl has done since version 4.000. Now system utilities that read the legacy process name such as ps, top and killall will recognize the name you set when assigning to `$0`. The string you supply will be cut off at 16 bytes, this is a limitation imposed by Linux.

Mnemonic: same as **sh** and **ksh**.

$REAL_GROUP_ID

$GID

$(

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

However, a value assigned to `$(` must be a single number used to set the real gid. So the value given by `$(` should *not* be assigned back to `$(` without being forced numeric, such as by adding zero. Note that this is different to the effective gid (`$)`) which does take a list.

You can change both the real gid and the effective gid at the same time by using `POSIX::setgid()`. Changes to `$(` require a check to `$!` to detect any possible errors after an attempted change.

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

$EFFECTIVE_GROUP_ID

$EGID

$)

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

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

You can change both the effective gid and the real gid at the same time by using `POSIX::setgid()` (use only a single numeric argument). Changes to `$)` require a check to `$!` to detect any possible errors after an attempted change.

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

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

$REAL_USER_ID

$UID

$<

The real uid of this process. You can change both the real uid and the effective uid at

the same time by using `POSIX::setuid()`. Since changes to $< require a system call, check `$!` after a change attempt to detect any possible errors.

Mnemonic: it's the uid you came *from*, if you're running setuid.

$EFFECTIVE_USER_ID

$EUID

$>

The effective uid of this process. For example:

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

You can change both the effective uid and the real uid at the same time by using `POSIX::setuid()`. Changes to $> require a check to `$!` to detect any possible errors after an attempted change.

$< and $> can be swapped only on machines supporting `setreuid()`.

Mnemonic: it's the uid you went *to*, if you're running setuid.

$SUBSCRIPT_SEPARATOR

$SUBSEP

$;

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

```
$foo{$x,$y,$z}
```

it really means

```
$foo{join($;, $x, $y, $z)}
```

But don't put

```
@foo{$x,$y,$z} # a slice--note the @
```

which means

```
($foo{$x},$foo{$y},$foo{$z})
```

Default is "\034", the same as SUBSEP in **awk**. If your keys contain binary data there might not be any safe value for `$;`.

Consider using "real" multidimensional arrays as described in *perllol*.

Mnemonic: comma (the syntactic subscript separator) is a semi-semicolon.

$a

$b

Special package variables when using `sort()`, see *"sort" in perlfunc*. Because of this specialness $a and $b don't need to be declared (using `use vars`, or `our()`) even when using the `strict 'vars'` pragma. Don't lexicalize them with `my $a` or `my $b` if you want to be able to use them in the `sort()` comparison block or function.

%ENV

The hash `%ENV` contains your current environment. Setting a value in `ENV` changes the environment for any child processes you subsequently `fork()` off.

As of v5.18.0, both keys and values stored in `%ENV` are stringified.

```
my $foo = 1;
```

```
$ENV{'bar'} = \$foo;
if( ref $ENV{'bar'} ) {
    say "Pre 5.18.0 Behaviour";
} else {
    say "Post 5.18.0 Behaviour";
}
```

Previously, only child processes received stringified values:

```
my $foo = 1;
$ENV{'bar'} = \$foo;

# Always printed 'non ref'
system($^X, '-e',
       q/print ( ref $ENV{'bar'}  ? 'ref' : 'non ref' ) /);
```

This happens because you can't really share arbitrary data structures with foreign processes.

$OLD_PERL_VERSION

$]

The revision, version, and subversion of the Perl interpreter, represented as a decimal of the form 5.XXXYYY, where XXX is the version / 1e3 and YYY is the subversion / 1e6. For example, Perl v5.10.1 would be "5.010001".

This variable can be used to determine whether the Perl interpreter executing a script is in the right range of versions:

```
warn "No PerlIO!\n" if $] lt '5.008';
```

When comparing `$]`, string comparison operators are **highly recommended**. The inherent limitations of binary floating point representation can sometimes lead to incorrect comparisons for some numbers on some architectures.

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

See *$^V* for a representation of the Perl version as a *version* object, which allows more flexible string comparisons.

The main advantage of `$]` over `$^V` is that it works the same on any version of Perl. The disadvantages are that it can't easily be compared to versions in other formats (e.g. literal v-strings, "v1.2.3" or version objects) and numeric comparisons can occasionally fail; it's good for string literal version checks and bad for comparing to a variable that hasn't been sanity-checked.

The `$OLD_PERL_VERSION` form was added in Perl v5.20.0 for historical reasons but its use is discouraged. (If your reason to use `$]` is to run code on old perls then referring to it as `$OLD_PERL_VERSION` would be self-defeating.)

Mnemonic: Is this version of perl in the right bracket?

$SYSTEM_FD_MAX

$^F

The maximum system file descriptor, ordinarily 2. System file descriptors are passed to `exec()`ed processes, while higher file descriptors are not. Also, during an `open()`, system file descriptors are preserved even if the `open()` fails (ordinary file descriptors are closed before the `open()` is attempted). The close-on-exec status of a file descriptor will be decided according to the value of `$^F` when the corresponding file, pipe, or socket was opened, not the time of the `exec()`.

@F

The array `@F` contains the fields of each line read in when autosplit mode is turned on. See *perlrun* for the **-a** switch. This array is package-specific, and must be declared or given a full package name if not in package main when running under `strict 'vars'`.

@INC

The array `@INC` contains the list of places that the `do EXPR`, `require`, or `use` constructs look for their library files. It initially consists of the arguments to any **-I** command-line switches, followed by the default Perl library, probably */usr/local/lib/perl*, followed by ".", to represent the current directory. ("." will not be appended if taint checks are enabled, either by `-T` or by `-t`.) If you need to modify this at runtime, you should use the `use lib` pragma to get the machine-dependent library properly loaded also:

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

You can also insert hooks into the file inclusion system by putting Perl code directly into `@INC`. Those hooks may be subroutine references, array references or blessed objects. See *"require" in perlfunc* for details.

%INC

The hash `%INC` contains entries for each filename included via the `do`, `require`, or `use` operators. The key is the filename you specified (with module names converted to pathnames), and the value is the location of the file found. The `require` operator uses this hash to determine whether a particular file has already been included.

If the file was loaded via a hook (e.g. a subroutine reference, see *"require" in perlfunc* for a description of these hooks), this hook is by default inserted into `%INC` in place of a filename. Note, however, that the hook may have set the `%INC` entry by itself to provide some more specific info.

$INPLACE_EDIT

$^I

The current value of the inplace-edit extension. Use `undef` to disable inplace editing.

Mnemonic: value of **-i** switch.

$^M

By default, running out of memory is an untrappable, fatal error. However, if suitably built, Perl can use the contents of `$^M` as an emergency memory pool after `die()`ing. Suppose that your Perl were compiled with `-DPERL_EMERGENCY_SBRK` and used Perl's malloc. Then

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

would allocate a 64K buffer for use in an emergency. See the *INSTALL* file in the Perl distribution for information on how to add custom C compilation flags when compiling perl. To discourage casual use of this advanced feature, there is no *English* long name for this variable.

This variable was added in Perl 5.004.

$OSNAME

$^O

The name of the operating system under which this copy of Perl was built, as determined during the configuration process. For examples see *"PLATFORMS" in*

perlport.

The value is identical to `$Config{'osname'}`. See also *Config* and the **-V** command-line switch documented in *perlrun*.

In Windows platforms, `$^O` is not very helpful: since it is always `MSWin32`, it doesn't tell the difference between 95/98/ME/NT/2000/XP/CE/.NET. Use `Win32::GetOSName()` or Win32::GetOSVersion() (see *Win32* and *perlport*) to distinguish between the variants.

This variable was added in Perl 5.003.

%SIG

The hash `%SIG` contains signal handlers for signals. For example:

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

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

Using a value of `'IGNORE'` usually has the effect of ignoring the signal, except for the `CHLD` signal. See *perlipc* for more about this special case.

Here are some other examples:

```
    $SIG{"PIPE"} = "Plumber";   # assumes main::Plumber (not
    # recommended)
    $SIG{"PIPE"} = \&Plumber;   # just fine; assume current
    # Plumber
    $SIG{"PIPE"} = *Plumber;    # somewhat esoteric
    $SIG{"PIPE"} = Plumber();   # oops, what did Plumber()
    # return??
```

Be sure not to use a bareword as the name of a signal handler, lest you inadvertently call it.

If your system has the `sigaction()` function then signal handlers are installed using it. This means you get reliable signal handling.

The default delivery policy of signals changed in Perl v5.8.0 from immediate (also known as "unsafe") to deferred, also known as "safe signals". See *perlipc* for more information.

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

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

As the `'IGNORE'` hook is not supported by `__WARN__`, you can disable warnings using the empty subroutine:

```
local $SIG{__WARN__} = sub {};
```

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

Due to an implementation glitch, the `$SIG{__DIE__}` hook is called even inside an `eval()`. Do not use this to rewrite a pending exception in `$@`, or as a bizarre substitute for overriding `CORE::GLOBAL::die()`. This strange action at a distance may be fixed in a future release so that `$SIG{__DIE__}` is only called if your program is about to exit, as was the original intent. Any other use is deprecated.

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

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

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

Having to even think about the `$^S` variable in your exception handlers is simply wrong. `$SIG{__DIE__}` as currently implemented invites grievous and difficult to track down errors. Avoid it and use an `END{}` or CORE::GLOBAL::die override instead.

See *"die" in perlfunc*, *"warn" in perlfunc*, *"eval" in perlfunc*, and *warnings* for additional information.

$BASETIME

$^T

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

$PERL_VERSION

$^V

The revision, version, and subversion of the Perl interpreter, represented as a *version* object.

This variable first appeared in perl v5.6.0; earlier versions of perl will see an undefined value. Before perl v5.10.0 `$^V` was represented as a v-string rather than a *version* object.

`$^V` can be used to determine whether the Perl interpreter executing a script is in the right range of versions. For example:

```
warn "Hashes not randomized!\n" if !$^V or $^V lt v5.8.1
```

While version objects overload stringification, to portably convert `$^V` into its string representation, use `sprintf()`'s `"%vd"` conversion, which works for both v-strings or version objects:

```
printf "version is v%vd\n", $^V;  # Perl's version
```

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

See also `$]` for a decimal representation of the Perl version.

The main advantage of `$^V` over `$]` is that, for Perl v5.10.0 or later, it overloads operators, allowing easy comparison against other version representations (e.g. decimal, literal v-string, "v1.2.3", or objects). The disadvantage is that prior to v5.10.0, it was only a literal v-string, which can't be easily printed or compared.

Mnemonic: use ^V for a version object.

${^WIN32_SLOPPY_STAT}

If this variable is set to a true value, then `stat()` on Windows will not try to open the file. This means that the link count cannot be determined and file attributes may be out of date if additional hardlinks to the file exist. On the other hand, not opening the file is considerably faster, especially for files on network drives.

This variable could be set in the *sitecustomize.pl* file to configure the local Perl installation to use "sloppy" `stat()` by default. See the documentation for **-f** in *perlrun* for more information about site customization.

This variable was added in Perl v5.10.0.

$EXECUTABLE_NAME

$^X

The name used to execute the current copy of Perl, from C's `argv[0]` or (where supported) */proc/self/exe*.

Depending on the host operating system, the value of `$^X` may be a relative or absolute pathname of the perl program file, or may be the string used to invoke perl but not the pathname of the perl program file. Also, most operating systems permit invoking programs that are not in the PATH environment variable, so there is no guarantee that the value of `$^X` is in PATH. For VMS, the value may or may not include a version number.

You usually can use the value of `$^X` to re-invoke an independent copy of the same perl that is currently running, e.g.,

```
@first_run = `$^X -le "print int rand 100 for 1..100"`;
```

But recall that not all operating systems support forking or capturing of the output of commands, so this complex statement may not be portable.

It is not safe to use the value of `$^X` as a path name of a file, as some operating systems that have a mandatory suffix on executable files do not require use of the suffix when invoking a command. To convert the value of `$^X` to a path name, use the following statements:

```
    # Build up a set of file names (not command names).
    use Config;
    my $this_perl = $^X;
    if ($^O ne 'VMS') {
 $this_perl .= $Config{_exe}
   unless $this_perl =~ m/$Config{_exe}$/i;
 }
```

Because many operating systems permit anyone with read access to the Perl program file to make a copy of it, patch the copy, and then execute the copy, the security-conscious Perl programmer should take care to invoke the installed copy of

perl, not the copy referenced by `$^X`. The following statements accomplish this goal, and produce a pathname that can be invoked as a command or referenced as a file.

```
    use Config;
    my $secure_perl_path = $Config{perlpath};
    if ($^O ne 'VMS') {
 $secure_perl_path .= $Config{_exe}
     unless $secure_perl_path =~ m/$Config{_exe}$/i;
 }
```

Variables related to regular expressions

Most of the special variables related to regular expressions are side effects. Perl sets these variables when it has a successful match, so you should check the match result before using them. For instance:

```
    if( /P(A)TT(ER)N/ ) {
 print "I found $1 and $2\n";
 }
```

These variables are read-only and dynamically-scoped, unless we note otherwise.

The dynamic nature of the regular expression variables means that their value is limited to the block that they are in, as demonstrated by this bit of code:

```
    my $outer = 'Wallace and Grommit';
    my $inner = 'Mutt and Jeff';

    my $pattern = qr/(\S+) and (\S+)/;

    sub show_n { print "\$1 is $1; \$2 is $2\n" }

    {
    OUTER:
 show_n() if $outer =~ m/$pattern/;

 INNER: {
     show_n() if $inner =~ m/$pattern/;
     }

 show_n();
    }
```

The output shows that while in the `OUTER` block, the values of `$1` and `$2` are from the match against `$outer`. Inside the `INNER` block, the values of `$1` and `$2` are from the match against `$inner`, but only until the end of the block (i.e. the dynamic scope). After the `INNER` block completes, the values of `$1` and `$2` return to the values for the match against `$outer` even though we have not made another match:

```
    $1 is Wallace; $2 is Grommit
    $1 is Mutt; $2 is Jeff
    $1 is Wallace; $2 is Grommit
```

Performance issues

Traditionally in Perl, any use of any of the three variables `$``, `$&` or `$'` (or their `use English` equivalents) anywhere in the code, caused all subsequent successful pattern matches to make a copy of the matched string, in case the code might subsequently access one of those variables. This imposed a considerable performance penalty across the whole program, so generally the use of these variables has been discouraged.

In Perl 5.6.0 the `@-` and `@+` dynamic arrays were introduced that supply the indices of successful matches. So you could for example do this:

```
    $str =~ /pattern/;

    print $`, $&, $'; # bad: perfomance hit

    print             # good: no perfomance hit
 substr($str, 0,     $-[0]),
 substr($str, $-[0], $+[0]-$-[0]),
 substr($str, $+[0]);
```

In Perl 5.10.0 the `/p` match operator flag and the `${^PREMATCH}`, `${^MATCH}`, and `${^POSTMATCH}` variables were introduced, that allowed you to suffer the penalties only on patterns marked with `/p`.

In Perl 5.18.0 onwards, perl started noting the presence of each of the three variables separately, and only copied that part of the string required; so in

```
    $`; $&; "abcdefgh" =~ /d/
```

perl would only copy the "abcd" part of the string. That could make a big difference in something like

```
    $str = 'x' x 1_000_000;
    $&; # whoops
    $str =~ /x/g # one char copied a million times, not a million chars
```

In Perl 5.20.0 a new copy-on-write system was enabled by default, which finally fixes all performance issues with these three variables, and makes them safe to use anywhere.

The `Devel::NYTProf` and `Devel::FindAmpersand` modules can help you find uses of these problematic match variables in your code.

$<*digits*> ($1, $2, ...)

Contains the subpattern from the corresponding set of capturing parentheses from the last successful pattern match, not counting patterns matched in nested blocks that have been exited already.

These variables are read-only and dynamically-scoped.

Mnemonic: like \digits.

$MATCH

$&

The string matched by the last successful pattern match (not counting any matches hidden within a BLOCK or `eval()` enclosed by the current BLOCK).

See *Performance issues* above for the serious performance implications of using this variable (even once) in your code.

This variable is read-only and dynamically-scoped.

Mnemonic: like & in some editors.

${^MATCH}

This is similar to $& ($MATCH) except that it does not incur the performance penalty associated with that variable.

See *Performance issues* above.

In Perl v5.18 and earlier, it is only guaranteed to return a defined value when the pattern was compiled or executed with the /p modifier. In Perl v5.20, the /p modifier does nothing, so ${^MATCH} does the same thing as $MATCH.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

$PREMATCH

$`

The string preceding whatever was matched by the last successful pattern match, not counting any matches hidden within a BLOCK or eval enclosed by the current BLOCK.

See *Performance issues* above for the serious performance implications of using this variable (even once) in your code.

This variable is read-only and dynamically-scoped.

Mnemonic: ` often precedes a quoted string.

${^PREMATCH}

This is similar to $` ($PREMATCH) except that it does not incur the performance penalty associated with that variable.

See *Performance issues* above.

In Perl v5.18 and earlier, it is only guaranteed to return a defined value when the pattern was compiled or executed with the /p modifier. In Perl v5.20, the /p modifier does nothing, so ${^PREMATCH} does the same thing as $PREMATCH.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

$POSTMATCH

$'

The string following whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or eval() enclosed by the current BLOCK). Example:

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

See *Performance issues* above for the serious performance implications of using this variable (even once) in your code.

This variable is read-only and dynamically-scoped.

Mnemonic: ' often follows a quoted string.

${^POSTMATCH}

This is similar to $' ($POSTMATCH) except that it does not incur the performance penalty associated with that variable.

See *Performance issues* above.

In Perl v5.18 and earlier, it is only guaranteed to return a defined value when the pattern was compiled or executed with the `/p` modifier. In Perl v5.20, the `/p` modifier does nothing, so `${^POSTMATCH}` does the same thing as `$POSTMATCH`.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

$LAST_PAREN_MATCH

$+

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

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

This variable is read-only and dynamically-scoped.

Mnemonic: be positive and forward looking.

$LAST_SUBMATCH_RESULT

$^N

The text matched by the used group most-recently closed (i.e. the group with the rightmost closing parenthesis) of the last successful search pattern.

This is primarily used inside `(?{...})` blocks for examining text recently matched. For example, to effectively capture text to a variable (in addition to `$1`, `$2`, etc.), replace `(...)` with

```
(?:(...)(?{ $var = $^N }))
```

By setting and then using `$var` in this way relieves you from having to worry about exactly which numbered set of parentheses they are.

This variable was added in Perl v5.8.0.

Mnemonic: the (possibly) Nested parenthesis that most recently closed.

@LAST_MATCH_END

@+

This array holds the offsets of the ends of the last successful submatches in the currently active dynamic scope. `$+[0]` is the offset into the string of the end of the entire match. This is the same value as what the `pos` function returns when called on the variable that was matched against. The *n*th element of this array holds the offset of the *n*th submatch, so `$+[1]` is the offset past where `$1` ends, `$+[2]` the offset past where `$2` ends, and so on. You can use `$#+` to determine how many subgroups were in the last successful match. See the examples given for the `@-` variable.

This variable was added in Perl v5.6.0.

%LAST_PAREN_MATCH

%+

Similar to `@+`, the `%+` hash allows access to the named capture buffers, should they exist, in the last successful match in the currently active dynamic scope.

For example, `$+{foo}` is equivalent to `$1` after the following match:

```
'foo' =~ /(?<foo>foo)/;
```

The keys of the `%+` hash list only the names of buffers that have captured (and that are thus associated to defined values).

The underlying behaviour of `%+` is provided by the *Tie::Hash::NamedCapture* module.

Note: `%-` and `%+` are tied views into a common internal hash associated with the last successful regular expression. Therefore mixing iterative access to them via `each` may have unpredictable results. Likewise, if the last successful match changes, then the results may be surprising.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

@LAST_MATCH_START

@-

`$-[0]` is the offset of the start of the last successful match. `$-[n]` is the offset of the start of the substring matched by *n*-th subpattern, or undef if the subpattern did not match.

Thus, after a match against `$_`, `$&` coincides with `substr $_, $-[0], $+[0] - $-[0]`. Similarly, `$n` coincides with `substr $_, $-[n], $+[n] - $-[n]` if `$-[n]` is defined, and `$+` coincides with `substr $_, $-[$#-], $+[$#-] - $-[$#-]`. One can use `$#-` to find the last matched subgroup in the last successful match. Contrast with `$#+`, the number of subgroups in the regular expression. Compare with `@+`.

This array holds the offsets of the beginnings of the last successful submatches in the currently active dynamic scope. `$-[0]` is the offset into the string of the beginning of the entire match. The *n*th element of this array holds the offset of the *n*th submatch, so `$-[1]` is the offset where `$1` begins, `$-[2]` the offset where `$2` begins, and so on.

After a match against some variable `$var`:

`$` ` is the same as `substr($var, 0, $-[0])`

`$&` is the same as `substr($var, $-[0], $+[0] - $-[0])`

`$'` is the same as `substr($var, $+[0])`

`$1` is the same as `substr($var, $-[1], $+[1] - $-[1])`

`$2` is the same as `substr($var, $-[2], $+[2] - $-[2])`

`$3` is the same as `substr($var, $-[3], $+[3] - $-[3])`

This variable was added in Perl v5.6.0.

%-

Similar to `%+`, this variable allows access to the named capture groups in the last successful match in the currently active dynamic scope. To each capture group name found in the regular expression, it associates a reference to an array containing the list of values captured by all buffers with that name (should there be several of them), in the order where they appear.

Here's an example:

```
if ('1234' =~ /(?<A>1)(?<B>2)(?<A>3)(?<B>4)/) {
    foreach my $bufname (sort keys %-) {
        my $ary = $-{$bufname};
        foreach my $idx (0..$#$ary) {
            print "\$-{$bufname}[$idx] : ",
                  (defined($ary->[$idx])
                      ? "'$ary->[$idx]'"
                      : "undef"),
                  "\n";
        }
    }
}
```

would print out:

```
$-{A}[0] : '1'
$-{A}[1] : '3'
$-{B}[0] : '2'
$-{B}[1] : '4'
```

The keys of the `%-` hash correspond to all buffer names found in the regular expression.

The behaviour of `%-` is implemented via the *Tie::Hash::NamedCapture* module.

Note: `%-` and `%+` are tied views into a common internal hash associated with the last successful regular expression. Therefore mixing iterative access to them via `each` may have unpredictable results. Likewise, if the last successful match changes, then the results may be surprising.

This variable was added in Perl v5.10.0.

This variable is read-only and dynamically-scoped.

$LAST_REGEXP_CODE_RESULT

$^R

The result of evaluation of the last successful `(?{ code })` regular expression assertion (see *perlre*). May be written to.

This variable was added in Perl 5.005.

${^RE_DEBUG_FLAGS}

The current value of the regex debugging flags. Set to 0 for no debug output even when the `re 'debug'` module is loaded. See *re* for details.

This variable was added in Perl v5.10.0.

${^RE_TRIE_MAXBUF}

Controls how certain regex optimisations are applied and how much memory they utilize. This value by default is 65536 which corresponds to a 512kB temporary cache. Set this to a higher value to trade memory for speed when matching large alternations. Set it to a lower value if you want the optimisations to be as conservative of memory as possible but still occur, and set it to a negative value to prevent the optimisation and conserve the most memory. Under normal situations this variable should be of no interest to you.

This variable was added in Perl v5.10.0.

Variables related to filehandles

Variables that depend on the currently selected filehandle may be set by calling an appropriate object method on the `IO::Handle` object, although this is less efficient than using the regular built-in variables. (Summary lines below for this contain the word HANDLE.) First you must say

```
use IO::Handle;
```

after which you may use either

```
method HANDLE EXPR
```

or more safely,

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

Each method returns the old value of the `IO::Handle` attribute. The methods each take an optional

EXPR, which, if supplied, specifies the new value for the `IO::Handle` attribute in question. If not supplied, most methods do nothing to the current value--except for `autoflush()`, which will assume a 1 for you, just to be different.

Because loading in the `IO::Handle` class is an expensive operation, you should learn how to use the regular built-in variables.

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

You should be very careful when modifying the default values of most special variables described in this document. In most cases you want to localize these variables before changing them, since if you don't, the change may affect other modules which rely on the default values of the special variables that you have changed. This is one of the correct ways to read the whole file at once:

```
open my $fh, "<", "foo" or die $!;
local $/; # enable localized slurp mode
my $content = <$fh>;
close $fh;
```

But the following code is quite bad:

```
open my $fh, "<", "foo" or die $!;
undef $/; # enable slurp mode
my $content = <$fh>;
close $fh;
```

since some other module, may want to read data from some file in the default "line mode", so if the code we have just presented has been executed, the global value of `$/` is now changed for any other code running inside the same Perl interpreter.

Usually when a variable is localized you want to make sure that this change affects the shortest scope possible. So unless you are already inside some short `{}` block, you should create one yourself. For example:

```
    my $content = '';
    open my $fh, "<", "foo" or die $!;
    {
 local $/;
 $content = <$fh>;
    }
    close $fh;
```

Here is an example of how your own code can go broken:

```
    for ( 1..3 ){
 $\ = "\r\n";
 nasty_break();
 print "$_";
    }

    sub nasty_break {
 $\ = "\f";
 # do something with $_
    }
```

You probably expect this code to print the equivalent of

```
"1\r\n2\r\n3\r\n"
```

but instead you get:

```
"1\f2\f3\f"
```

Why? Because `nasty_break()` modifies `$\` without localizing it first. The value you set in `nasty_break()` is still there when you return. The fix is to add `local()` so the value doesn't leak out of `nasty_break()`:

```
local $\ = "\f";
```

It's easy to notice the problem in such a short example, but in more complicated code you are looking for trouble if you don't localize changes to the special variables.

$ARGV

Contains the name of the current file when reading from <>.

@ARGV

The array `@ARGV` contains the command-line arguments intended for the script. `$#ARGV` is generally the number of arguments minus one, because `$ARGV[0]` is the first argument, *not* the program's command name itself. See *$0* for the command name.

ARGV

The special filehandle that iterates over command-line filenames in `@ARGV`. Usually written as the null filehandle in the angle operator <>. Note that currently `ARGV` only has its magical effect within the <> operator; elsewhere it is just a plain filehandle corresponding to the last file opened by <>. In particular, passing `\*ARGV` as a parameter to a function that expects a filehandle may not cause your function to automatically read the contents of all the files in `@ARGV`.

ARGVOUT

The special filehandle that points to the currently open output file when doing edit-in-place processing with **-i**. Useful when you have to do a lot of inserting and don't want to keep modifying $_. See *perlrun* for the **-i** switch.

IO::Handle->output_field_separator(EXPR)

$OUTPUT_FIELD_SEPARATOR

$OFS

$,

The output field separator for the print operator. If defined, this value is printed between each of print's arguments. Default is `undef`.

You cannot call `output_field_separator()` on a handle, only as a static method. See *IO::Handle*.

Mnemonic: what is printed when there is a "," in your print statement.

HANDLE->input_line_number(EXPR)

$INPUT_LINE_NUMBER

$NR

$.

Current line number for the last filehandle accessed.

Each filehandle in Perl counts the number of lines that have been read from it.

(Depending on the value of `$/`, Perl's idea of what constitutes a line may not match yours.) When a line is read from a filehandle (via `readline()` or <>), or when `tell()` or `seek()` is called on it, `$.` becomes an alias to the line counter for that filehandle.

You can adjust the counter by assigning to `$.`, but this will not actually move the seek pointer. *Localizing `$.` will not localize the filehandle's line count*. Instead, it will localize perl's notion of which filehandle `$.` is currently aliased to.

`$.` is reset when the filehandle is closed, but **not** when an open filehandle is reopened without an intervening `close()`. For more details, see *"I/O Operators" in perlop*. Because <> never does an explicit close, line numbers increase across `ARGV` files (but see examples in *"eof" in perlfunc*).

You can also use `HANDLE->input_line_number(EXPR)` to access the line counter for a given filehandle without having to worry about which handle you last accessed.

Mnemonic: many programs use "." to mean the current line number.

IO::Handle->input_record_separator(EXPR)

$INPUT_RECORD_SEPARATOR

$RS

$/

The input record separator, newline by default. This influences Perl's idea of what a "line" is. Works like **awk**'s RS variable, including treating empty lines as a terminator if set to the null string (an empty line cannot contain any spaces or tabs). You may set it to a multi-character string to match a multi-character terminator, or to `undef` to read through the end of file. Setting it to `"\n\n"` means something slightly different than setting to `""`, if the file contains consecutive empty lines. Setting to `""` will treat two or more consecutive empty lines as a single empty line. Setting to `"\n\n"` will blindly assume that the next input character belongs to the next paragraph, even if it's a newline.

```
local $/;           # enable "slurp" mode
local $_ = <FH>;    # whole file now here
s/\n[ \t]+/ /g;
```

Remember: the value of `$/` is a string, not a regex. **awk** has to be better for something. :-)

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

```
local $/ = \32768; # or \"32768", or \$var_containing_32768
open my $fh, "<", $myfile or die $!;
local $_ = <$fh>;
```

will read a record of no more than 32768 characters from $fh. If you're not reading from a record-oriented file (or your OS doesn't have record-oriented files), then you'll likely get a full chunk of data with every read. If a record is larger than the record size you've set, you'll get the record back in pieces. Trying to set the record size to zero or less is deprecated and will cause $/ to have the value of "undef", which will cause reading in the (rest of the) whole file.

As of 5.19.9 setting `$/` to any other form of reference will throw a fatal exception. This is in preparation for supporting new ways to set `$/` in the future.

On VMS only, record reads bypass PerlIO layers and any associated buffering, so you must not mix record and non-record reads on the same filehandle. Record mode mixes with line mode only when the same buffering layer is in use for both modes.

You cannot call `input_record_separator()` on a handle, only as a static method. See *IO::Handle*.

See also *"Newlines" in perlport*. Also see *$.*.

Mnemonic: / delimits line boundaries when quoting poetry.

IO::Handle->output_record_separator(EXPR)

$OUTPUT_RECORD_SEPARATOR

$ORS

$\

The output record separator for the print operator. If defined, this value is printed after the last of print's arguments. Default is `undef`.

You cannot call `output_record_separator()` on a handle, only as a static method. See *IO::Handle*.

Mnemonic: you set `$\` instead of adding "\n" at the end of the print. Also, it's just like `$/`, but it's what you get "back" from Perl.

HANDLE->autoflush(EXPR)

$OUTPUT_AUTOFLUSH

$|

If set to nonzero, forces a flush right away and after every write or print on the currently selected output channel. Default is 0 (regardless of whether the channel is really buffered by the system or not; `$|` tells you only whether you've asked Perl explicitly to flush after each write). STDOUT will typically be line buffered if output is to the terminal and block buffered otherwise. Setting this variable is useful primarily when you are outputting to a pipe or socket, such as when you are running a Perl program under **rsh** and want to see the output as it's happening. This has no effect on input buffering. See *"getc" in perlfunc* for that. See *"select" in perlfunc* on how to select the output channel. See also *IO::Handle*.

Mnemonic: when you want your pipes to be piping hot.

${^LAST_FH}

This read-only variable contains a reference to the last-read filehandle. This is set by `<HANDLE>`, `readline`, `tell`, `eof` and `seek`. This is the same handle that `$.` and `tell` and `eof` without arguments use. It is also the handle used when Perl appends ", <STDIN> line 1" to an error or warning message.

This variable was added in Perl v5.18.0.

Variables related to formats

The special variables for formats are a subset of those for filehandles. See *perlform* for more information about Perl's formats.

$ACCUMULATOR

$^A

The current value of the `write()` accumulator for `format()` lines. A format contains `formline()` calls that put their result into `$^A`. After calling its format, `write()` prints out the contents of `$^A` and empties. So you never really see the contents of `$^A` unless you call `formline()` yourself and then look at it. See *perlform* and *"formline PICTURE,LIST" in perlfunc*.

IO::Handle->format_formfeed(EXPR)

$FORMAT_FORMFEED

$^L

What formats output as a form feed. The default is `\f`.

You cannot call `format_formfeed()` on a handle, only as a static method. See *IO::Handle*.

HANDLE->format_page_number(EXPR)

$FORMAT_PAGE_NUMBER

$%

The current page number of the currently selected output channel.

Mnemonic: % is page number in **nroff**.

HANDLE->format_lines_left(EXPR)

$FORMAT_LINES_LEFT

$-

The number of lines left on the page of the currently selected output channel.

Mnemonic: lines_on_page - lines_printed.

IO::Handle->format_line_break_characters EXPR

$FORMAT_LINE_BREAK_CHARACTERS

$:

The current set of characters after which a string may be broken to fill continuation fields (starting with ^) in a format. The default is " \n-", to break on a space, newline, or a hyphen.

You cannot call `format_line_break_characters()` on a handle, only as a static method. See *IO::Handle*.

Mnemonic: a "colon" in poetry is a part of a line.

HANDLE->format_lines_per_page(EXPR)

$FORMAT_LINES_PER_PAGE

$=

The current page length (printable lines) of the currently selected output channel. The default is 60.

Mnemonic: = has horizontal lines.

HANDLE->format_top_name(EXPR)

$FORMAT_TOP_NAME

$^

The name of the current top-of-page format for the currently selected output channel. The default is the name of the filehandle with `_TOP` appended. For example, the default format top name for the `STDOUT` filehandle is `STDOUT_TOP`.

Mnemonic: points to top of page.

HANDLE->format_name(EXPR)

$FORMAT_NAME

$~

The name of the current report format for the currently selected output channel. The default format name is the same as the filehandle name. For example, the default format name for the `STDOUT` filehandle is just `STDOUT`.

Mnemonic: brother to $^.

Error Variables

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

To illustrate the differences between these variables, consider the following Perl expression, which uses a single-quoted string. After execution of this statement, perl may have set all four special error variables:

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

When perl executes the `eval()` expression, it translates the `open()`, `<PIPE>`, and `close` calls in the C run-time library and thence to the operating system kernel. perl sets `$!` to the C library's `errno` if one of these calls fails.

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

Under a few operating systems, `$^E` may contain a more verbose error indicator, such as in this case, "CDROM tray not closed." Systems that do not support extended error messages leave `$^E` the same as `$!`.

Finally, $? may be set to a non-0 value if the external program */cdrom/install* fails. The upper eight bits reflect specific error conditions encountered by the program (the program's `exit()` value). The lower eight bits reflect mode of failure, like signal death and core dump information. See *wait(2)* for details. In contrast to `$!` and `$^E`, which are set only if an error condition is detected, the variable $? is set on each `wait` or pipe `close`, overwriting the old value. This is more like $@, which on every `eval()` is always set on failure and cleared on success.

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

${^CHILD_ERROR_NATIVE}

The native status returned by the last pipe close, backtick (`` ``) command, successful call to `wait()` or `waitpid()`, or from the `system()` operator. On POSIX-like systems this value can be decoded with the WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG and WIFCONTINUED functions provided by the *POSIX* module.

Under VMS this reflects the actual VMS exit status; i.e. it is the same as $? when the pragma `use vmsish 'status'` is in effect.

This variable was added in Perl v5.10.0.

$EXTENDED_OS_ERROR

$^E

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

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

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

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

Caveats mentioned in the description of *$!* generally apply to $^E, also.

This variable was added in Perl 5.003.

Mnemonic: Extra error explanation.

$EXCEPTIONS_BEING_CAUGHT

$^S

Current state of the interpreter.

```
$^S         State
---------   ------------------------------------
undef       Parsing module, eval, or main program
true (1)    Executing an eval
false (0)   Otherwise
```

The first state may happen in `$SIG{__DIE__}` and `$SIG{__WARN__}` handlers.

The English name $EXCEPTIONS_BEING_CAUGHT is slightly misleading, because the `undef` value does not indicate whether exceptions are being caught, since compilation of the main program does not catch exceptions.

This variable was added in Perl 5.004.

$WARNING

$^W

The current value of the warning switch, initially true if **-w** was used, false otherwise, but directly modifiable.

See also *warnings*.

Mnemonic: related to the **-w** switch.

${^WARNING_BITS}

The current set of warning checks enabled by the `use warnings` pragma. It has the same scoping as the $^H and %^H variables. The exact values are considered internal to the *warnings* pragma and may change between versions of Perl.

This variable was added in Perl v5.6.0.

$OS_ERROR

$ERRNO

$!

When referenced, `$!` retrieves the current value of the C `errno` integer variable. If `$!` is assigned a numerical value, that value is stored in `errno`. When referenced as a string, `$!` yields the system error string corresponding to `errno`.

Many system or library calls set `errno` if they fail, to indicate the cause of failure. They usually do **not** set `errno` to zero if they succeed. This means `errno`, hence `$!`, is meaningful only *immediately* after a **failure**:

```
    if (open my $fh, "<", $filename) {
  # Here $! is meaningless.
  ...
    }
```

```
    else {
  # ONLY here is $! meaningful.
  ...
  # Already here $! might be meaningless.
    }
    # Since here we might have either success or failure,
    # $! is meaningless.
```

Here, *meaningless* means that `$!` may be unrelated to the outcome of the `open()` operator. Assignment to `$!` is similarly ephemeral. It can be used immediately before invoking the `die()` operator, to set the exit value, or to inspect the system error string corresponding to error *n*, or to restore `$!` to a meaningful state.

Mnemonic: What just went bang?

%OS_ERROR

%ERRNO

%!

Each element of `%!` has a true value only if `$!` is set to that value. For example, `$!{ENOENT}` is true if and only if the current value of `$!` is `ENOENT`; that is, if the most recent error was "No such file or directory" (or its moral equivalent: not all operating systems give that exact error, and certainly not all languages). The specific true value is not guaranteed, but in the past has generally been the numeric value of `$!`. To check if a particular key is meaningful on your system, use `exists $!{the_key}`; for a list of legal keys, use `keys %!`. See *Errno* for more information, and also see *$!*.

This variable was added in Perl 5.005.

$CHILD_ERROR

$?

The status returned by the last pipe close, backtick (`` ` ``) command, successful call to `wait()` or `waitpid()`, or from the `system()` operator. This is just the 16-bit status word returned by the traditional Unix `wait()` system call (or else is made up to look like it). Thus, the exit value of the subprocess is really (`$? >> 8`), and `$? & 127` gives which signal, if any, the process died from, and `$? & 128` reports whether there was a core dump.

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

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

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

```
    END {
 $? = 1 if $? == 255;  # die would make it 255
    }
```

Under VMS, the pragma `use vmsish 'status'` makes $? reflect the actual VMS exit status, instead of the default emulation of POSIX status; see *"$?" in perlvms* for details.

Mnemonic: similar to **sh** and **ksh**.

$EVAL_ERROR

$@

The Perl error from the last `eval` operator, i.e. the last exception that was caught. For `eval BLOCK`, this is either a runtime error message or the string or reference `die` was called with. The `eval STRING` form also catches syntax errors and other compile time exceptions.

If no error occurs, `eval` sets `$@` to the empty string.

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

Mnemonic: Where was the error "at"?

Variables related to the interpreter state

These variables provide information about the current interpreter state.

$COMPILING

$^C

The current value of the flag associated with the **-c** switch. Mainly of use with **-MO=...** to allow code to alter its behavior when being compiled, such as for example to `AUTOLOAD` at compile time rather than normal, deferred loading. Setting `$^C = 1` is similar to calling `B::minus_c`.

This variable was added in Perl v5.6.0.

$DEBUGGING

$^D

The current value of the debugging flags. May be read or set. Like its *command-line equivalent*, you can use numeric or symbolic values, e.g. `$^D = 10` or `$^D = "st"`. See *"**-D**number" in perlrun*. The contents of this variable also affects the debugger operation. See *"Debugger Internals" in perldebguts*.

Mnemonic: value of **-D** switch.

${^ENCODING}

DEPRECATED!!!

The *object reference* to the `Encode` object that is used to convert the source code to Unicode. Thanks to this variable your Perl script does not have to be written in UTF-8. Default is `undef`.

Setting this variable to any other value than `undef` is deprecated due to fundamental defects in its design and implementation. It is planned to remove it from a future Perl version. Its purpose was to allow your non-ASCII Perl scripts to not have to be written in UTF-8; this was useful before editors that worked on UTF-8 encoded text were common, but that was long ago. It causes problems, such as affecting the operation of other modules that aren't expecting it, causing general mayhem. Its use can lead to segfaults.

If you need something like this functionality, you should use the *encoding* pragma, which is also deprecated, but has fewer nasty side effects.

If you are coming here because code of yours is being adversely affected by someone's use of this variable, you can usually work around it by doing this:

```
local ${^ENCODING};
```

near the beginning of the functions that are getting broken. This undefines the variable during the scope of execution of the including function.

This variable was added in Perl 5.8.2.

${^GLOBAL_PHASE}

The current phase of the perl interpreter.

Possible values are:

CONSTRUCT

The `PerlInterpreter*` is being constructed via `perl_construct`. This value is mostly there for completeness and for use via the underlying C variable `PL_phase`. It's not really possible for Perl code to be executed unless construction of the interpreter is finished.

START

This is the global compile-time. That includes, basically, every `BEGIN` block executed directly or indirectly from during the compile-time of the top-level program.

This phase is not called "BEGIN" to avoid confusion with `BEGIN`-blocks, as those are executed during compile-time of any compilation unit, not just the top-level program. A new, localised compile-time entered at run-time, for example by constructs as `eval "use SomeModule"` are not global interpreter phases, and therefore aren't reflected by `${^GLOBAL_PHASE}`.

CHECK

Execution of any `CHECK` blocks.

INIT

Similar to "CHECK", but for `INIT`-blocks, not `CHECK` blocks.

RUN

The main run-time, i.e. the execution of `PL_main_root`.

END

Execution of any `END` blocks.

DESTRUCT

Global destruction.

Also note that there's no value for UNITCHECK-blocks. That's because those are run for each compilation unit individually, and therefore is not a global interpreter phase.

Not every program has to go through each of the possible phases, but transition from one phase to another can only happen in the order described in the above list.

An example of all of the phases Perl code can see:

```
BEGIN { print "compile-time: ${^GLOBAL_PHASE}\n" }

INIT  { print "init-time: ${^GLOBAL_PHASE}\n" }

CHECK { print "check-time: ${^GLOBAL_PHASE}\n" }

{
    package Print::Phase;

    sub new {
        my ($class, $time) = @_;
        return bless \$time, $class;
    }

    sub DESTROY {
        my $self = shift;
```

```
            print "$$self: ${^GLOBAL_PHASE}\n";
        }
    }

    print "run-time: ${^GLOBAL_PHASE}\n";

    my $runtime = Print::Phase->new(
        "lexical variables are garbage collected before END"
    );

    END   { print "end-time: ${^GLOBAL_PHASE}\n" }

    our $destruct = Print::Phase->new(
        "package variables are garbage collected after END"
    );
```

This will print out

```
compile-time: START
check-time: CHECK
init-time: INIT
run-time: RUN
lexical variables are garbage collected before END: RUN
end-time: END
package variables are garbage collected after END: DESTRUCT
```

This variable was added in Perl 5.14.0.

$^H

WARNING: This variable is strictly for internal use only. Its availability, behavior, and contents are subject to change without notice.

This variable contains compile-time hints for the Perl interpreter. At the end of compilation of a BLOCK the value of this variable is restored to the value when the interpreter started to compile the BLOCK.

When perl begins to parse any block construct that provides a lexical scope (e.g., eval body, required file, subroutine body, loop body, or conditional block), the existing value of `$^H` is saved, but its value is left unchanged. When the compilation of the block is completed, it regains the saved value. Between the points where its value is saved and restored, code that executes within BEGIN blocks is free to change the value of `$^H`.

This behavior provides the semantic of lexical scoping, and is used in, for instance, the `use strict` pragma.

The contents should be an integer; different bits of it are used for different pragmatic flags. Here's an example:

```
    sub add_100 { $^H |= 0x100 }

    sub foo {
 BEGIN { add_100() }
 bar->baz($boon);
    }
```

Consider what happens during execution of the BEGIN block. At this point the BEGIN block has already been compiled, but the body of `foo()` is still being compiled. The new value of `$^H` will therefore be visible only while the body of `foo()` is being compiled.

Substitution of `BEGIN { add_100() }` block with:

```
    BEGIN { require strict; strict->import('vars') }
```

demonstrates how `use strict 'vars'` is implemented. Here's a conditional version of the same lexical pragma:

```
    BEGIN {
 require strict; strict->import('vars') if $condition
    }
```

This variable was added in Perl 5.003.

%^H

The `%^H` hash provides the same scoping semantic as `$^H`. This makes it useful for implementation of lexically scoped pragmas. See *perlpragma*. All the entries are stringified when accessed at runtime, so only simple values can be accommodated. This means no pointers to objects, for example.

When putting items into `%^H`, in order to avoid conflicting with other users of the hash there is a convention regarding which keys to use. A module should use only keys that begin with the module's name (the name of its main package) and a "/" character. For example, a module `Foo::Bar` should use keys such as `Foo::Bar/baz`.

This variable was added in Perl v5.6.0.

${^OPEN}

An internal variable used by PerlIO. A string in two parts, separated by a `\0` byte, the first part describes the input layers, the second part describes the output layers.

This variable was added in Perl v5.8.0.

$PERLDB

$^P

The internal variable for debugging support. The meanings of the various bits are subject to change, but currently indicate:

0x01

Debug subroutine enter/exit.

0x02

Line-by-line debugging. Causes `DB::DB()` subroutine to be called for each statement executed. Also causes saving source code lines (like 0x400).

0x04

Switch off optimizations.

0x08

Preserve more data for future interactive inspections.

0x10

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

0x20

Start with single-step on.

0x40

Use subroutine address instead of name when reporting.

0x80

Report `goto &subroutine` as well.

0x100

Provide informative "file" names for evals based on the place they were compiled.

0x200

Provide informative names to anonymous subroutines based on the place they were compiled.

0x400

Save source code lines into `@{"_<$filename"}`.

0x800

When saving source, include evals that generate no subroutines.

0x1000

When saving source, include source that did not compile.

Some bits may be relevant at compile-time only, some at run-time only. This is a new mechanism and the details may change. See also *perldebguts*.

${^TAINT}

Reflects if taint mode is on or off. 1 for on (the program was run with **-T**), 0 for off, -1 when only taint warnings are enabled (i.e. with **-t** or **-TU**).

This variable is read-only.

This variable was added in Perl v5.8.0.

${^UNICODE}

Reflects certain Unicode settings of Perl. See *perlrun* documentation for the `-C` switch for more information about the possible values.

This variable is set during Perl startup and is thereafter read-only.

This variable was added in Perl v5.8.2.

${^UTF8CACHE}

This variable controls the state of the internal UTF-8 offset caching code. 1 for on (the default), 0 for off, -1 to debug the caching code by checking all its results against linear scans, and panicking on any discrepancy.

This variable was added in Perl v5.8.9. It is subject to change or removal without notice, but is currently used to avoid recalculating the boundaries of multi-byte UTF-8-encoded characters.

${^UTF8LOCALE}

This variable indicates whether a UTF-8 locale was detected by perl at startup. This information is used by perl when it's in adjust-utf8ness-to-locale mode (as when run with the `-CL` command-line switch); see *perlrun* for more info on this.

This variable was added in Perl v5.8.8.

Deprecated and removed variables

Deprecating a variable announces the intent of the perl maintainers to eventually remove the variable from the language. It may still be available despite its status. Using a deprecated variable triggers a warning.

Once a variable is removed, its use triggers an error telling you the variable is unsupported.

See *perldiag* for details about error messages.

$#

$# was a variable that could be used to format printed numbers. After a deprecation cycle, its magic was removed in Perl v5.10.0 and using it now triggers a warning: `$# is no longer supported`.

This is not the sigil you use in front of an array name to get the last index, like `$#array`. That's still how you get the last index of an array in Perl. The two have nothing to do with each other.

Deprecated in Perl 5.

Removed in Perl v5.10.0.

$*

`$*` was a variable that you could use to enable multiline matching. After a deprecation cycle, its magic was removed in Perl v5.10.0. Using it now triggers a warning: `$* is no longer supported`. You should use the `/s` and `/m` regexp modifiers instead.

Deprecated in Perl 5.

Removed in Perl v5.10.0.

$[

This variable stores the index of the first element in an array, and of the first character in a substring. The default is 0, but you could theoretically set it to 1 to make Perl behave more like **awk** (or Fortran) when subscripting and when evaluating the index() and substr() functions.

As of release 5 of Perl, assignment to `$[` is treated as a compiler directive, and cannot influence the behavior of any other file. (That's why you can only assign compile-time constants to it.) Its use is highly discouraged.

Prior to Perl v5.10.0, assignment to `$[` could be seen from outer lexical scopes in the same file, unlike other compile-time directives (such as *strict*). Using local() on it would bind its value strictly to a lexical block. Now it is always lexically scoped.

As of Perl v5.16.0, it is implemented by the *arybase* module. See *arybase* for more details on its behaviour.

Under `use v5.16`, or `no feature "array_base"`, `$[` no longer has any effect, and always contains 0. Assigning 0 to it is permitted, but any other value will produce an error.

Mnemonic: [begins subscripts.

Deprecated in Perl v5.12.0.