NAME

perlvar - Perl predefined variables

DESCRIPTION

Predefined Names

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**. In general, it's best to use the

```
use English '-no_match_vars';
```

invocation if you don't need $PREMATCH, $MATCH, or $POSTMATCH, as it avoids a certain performance hit with the use of regular expressions. See *English*.

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..5){
    nasty_break();
    print "$_ ";
}
sub nasty_break {
    $_ = 5;
    # do something with $_
}
```

You probably expect this code to print:

```
1 2 3 4 5
```

but instead you get:

```
5 5 5 5 5
```

Why? Because nasty_break() modifies $_ without localizing it first. The fix is to add local():

```
    local $_ = 5;
```

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.

The following list is ordered by scalar variables first, then the arrays, then the hashes.

$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:

  abs, alarm, chomp, chop, chr, chroot, cos, defined, eval, exp, glob, hex, int, lc, lcfirst, length, log, lstat, mkdir, oct, ord, pos, print, quotemeta, readlink, readpipe, ref, require, reverse (in scalar context only), rmdir, sin, split (on 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 an input record when a <FH> operation's result is tested by itself as the sole criterion of a `while` test. Outside a `while` test, this will not happen.

As $_ is a global variable, this may lead in some cases to unwanted side-effects. As of perl 5.9.1, you can now 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.

(Mnemonic: underline is understood in certain operations.)

$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.

$<*digits*>

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

$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 and dynamically scoped to the current BLOCK.

The use of this variable anywhere in a program imposes a considerable performance penalty on all regular expression matches. See *BUGS*.

See @- for a replacement.

${^MATCH}

This is similar to $& ($MATCH) except that it does not incur the performance penalty associated with that variable, and is only guaranteed to return a defined value when the pattern was compiled or executed with the /p modifier.

$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.

The use of this variable anywhere in a program imposes a considerable performance penalty on all regular expression matches. See *BUGS*.

See @- for a replacement.

${^PREMATCH}

This is similar to $` ($PREMATCH) except that it does not incur the performance penalty associated with that variable, and is only guaranteed to return a defined value when the pattern was compiled or executed with the /p modifier.

$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:

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

This variable is read-only and dynamically scoped to the current BLOCK.

The use of this variable anywhere in a program imposes a considerable performance penalty on all regular expression matches. See *BUGS*.

See @- for a replacement.

${^POSTMATCH}

This is similar to $' ($POSTMATCH) except that it does not incur the performance penalty associated with that variable, and is only guaranteed to return a defined value when the pattern was compiled or executed with the /p modifier.

$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 = $+);
```

(Mnemonic: be positive and forward looking.) This variable is read-only and dynamically scoped to the current BLOCK.

$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. (Mnemonic: the

(possibly) Nested parenthesis that most recently closed.)

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 is dynamically scoped to the current BLOCK.

@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.

%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.

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. (Mnemonic: / delimits line boundaries when quoting poetry.)

```
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. 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 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. Trying to set the record size to zero or less will cause reading in the (rest of the) whole file.

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 unlikely to be a problem, because any file you'd want to read in record mode is probably unusable in line mode.) Non-VMS systems do normal I/O, so it's safe to mix record and non-record reads of a file.

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

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 perldoc* on how to select the output channel. See also *IO::Handle*. (Mnemonic: when you want your pipes to be piping hot.)

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

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`. (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 and slice 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**. 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 as described in *perllol*.

HANDLE->format_page_number(EXPR)

$FORMAT_PAGE_NUMBER

$%

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

HANDLE->format_lines_per_page(EXPR)

$FORMAT_LINES_PER_PAGE

$=

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

HANDLE->format_lines_left(EXPR)

$FORMAT_LINES_LEFT

$-

The number of lines left on the page of the currently selected output channel. Used with formats. (Mnemonic: lines_on_page - lines_printed.)

@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])`

%-

Similar to %+, this variable allows access to the named capture buffers in the last successful match in the currently active dynamic scope. To each capture buffer 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.

HANDLE->format_name(EXPR)

$FORMAT_NAME

$~

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

HANDLE->format_top_name(EXPR)

$FORMAT_TOP_NAME

$^

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

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. Default is " \n-", to break on whitespace or hyphens. (Mnemonic: a "colon" in poetry is a part of a line.)

IO::Handle->format_formfeed EXPR

$FORMAT_FORMFEED

$^L

What formats output as a form feed. Default is \f.

$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()" in perlfunc*

$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. (Mnemonic: similar to **sh** and **ksh**.)

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.

Also see *Error Indicators*.

${^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.

${^ENCODING}

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*. The direct manipulation of this variable is highly discouraged.

$OS_ERROR

$ERRNO

$!

If used numerically, yields the current value of the C `errno` variable, or in other words, if a system or library call fails, it sets this variable. This means that the value of `$!` 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,
    # here $! is meaningless.
```

In the above *meaningless* stands for anything: zero, non-zero, `undef`. A successful system or library call does **not** set the variable to zero.

If used as a string, yields the corresponding system error string. You can assign a number 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 *Error Indicators*.

%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). To check if a particular key is meaningful on your system, use `exists $!{the_key}`; for a list of legal keys, use `keys %!`. See *Errno* for more information, and also see above for the validity of `$!`.

$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. (Mnemonic: Extra error explanation.)

Also see *Error Indicators*.

$EVAL_ERROR

$@

The Perl syntax error message from the last eval() operator. If $@ is the null string, 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"?)

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 *Error Indicators*.

$PROCESS_ID

$PID

$$

The process number of the Perl running this script. You should consider this variable read-only, although it will be altered across fork() calls. (Mnemonic: same as shells.)

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

$REAL_USER_ID

$UID

$<

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

$EFFECTIVE_USER_ID

$EUID

$>

The effective uid of this process. Example:

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

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.

(Mnemonic: it's the uid you went *to*, if you're running setuid.) $< 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 `$(` without being forced numeric, such as by adding zero. Note that this is different to the effective gid (`$)`) 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.

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

`$<`, `$>`, `$(` and `$)` 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 program being executed.

On some (read: 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. (Mnemonic: same as **sh** and **ksh**.)

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"`.

$[

The index of the first element in an array, and of the first character in a substring. 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. (Mnemonic: [begins subscripts.)

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.

Note that, unlike other compile-time directives (such as *strict*), assignment to `$[` can be seen from outer lexical scopes in the same file. However, you can use local() on it

to strictly bind its value to a lexical block.

$]

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 running Perl interpreter is too old.

The floating point representation can sometimes lead to inaccurate numeric comparisons. See `$^V` for a more modern representation of the Perl version that allows accurate string comparisons.

$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`.

$DEBUGGING

$^D

The current value of the debugging flags. (Mnemonic: value of **-D** switch.) May be read or set. Like its command-line equivalent, you can use numeric or symbolic values, eg `$^D = 10` or `$^D = "st"`.

${^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.

${^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.

$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().

$^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 the above BEGIN 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
 }
```

%^H

The %^H hash provides the same scoping semantic as $^H. This makes it useful for implementation of lexically scoped pragmas. See *perlpragma*.

$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.

$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'}`. 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.

${^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.

$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"}`.

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*.

$LAST_REGEXP_CODE_RESULT

$^R

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

$EXCEPTIONS_BEING_CAUGHT

$^S

Current state of the interpreter.

```
$^S         State
---------   -------------------
undef       Parsing module/eval
true (1)    Executing an eval
false (0)   Otherwise
```

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

$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.

${^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.

${^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.

${^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.

${^UTF8LOCALE}

This variable indicates whether an 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.

$PERL_VERSION

$^V

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

This variable first appeared in perl 5.6.0; earlier versions of perl will see an undefined value. Before perl 5.10.0 $^V was represented as a v-string.

$^V can be used to determine whether the Perl interpreter executing a script is in the right range of versions. (Mnemonic: use ^V for Version Control.) Example:

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

To convert $^V into its string representation use sprintf()'s "%vd" conversion:

```
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 an older representation of the Perl version.

$WARNING

$^W

The current value of the warning switch, initially true if **-w** was used, false otherwise, but directly modifiable. (Mnemonic: related to the **-w** switch.) See also *warnings*.

${^WARNING_BITS}

The current set of warning checks enabled by the `use warnings` pragma. See the documentation of `warnings` for more details.

${^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 *perlrun* for more information about site customization.

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

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.

$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.

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.

@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.

@ARG

@_

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

%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.

%ENV

$ENV{expr}

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

%SIG

$SIG{expr}

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 5.8.0 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, 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...
         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 *"die" in perlfunc*, *"warn" in perlfunc*, *"eval" in perlfunc*, and *warnings* for additional information.

Error Indicators

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:

```
    eval q{
open my $pipe, "/cdrom/install |" or die $!;
my @res = <$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 . In these cases the value of

$@ is the compile error, or the argument to `die` (which will interpolate `$!` and $?). (See also *Fatal*, though.)

When the eval() expression above is executed, open(), `<PIPE>`, and `close` are translated to calls in the C run-time library and thence to the operating system kernel. `$!` is set to the C library's `errno` if one of these calls fails.

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, $? may be set to non-0 value if the external program */cdrom/install* fails. The upper eight bits reflect specific error conditions encountered by the program (the program's exit() value). The lower eight bits reflect mode of failure, like signal death and core dump information See wait(2) for details. In contrast to `$!` and `$^E`, which are set only if 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 $?.

Technical Note on 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*.

Perl variable names may also be a sequence of digits or a single punctuation or control character. 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. Perl has a special syntax for the single-control-character names: It understands `^X` (caret `X`) to mean the control-`X` character. For example, the notation `$^W` (dollar-sign caret `W`) is the scalar variable whose name is the single character control-`W`. This is better than typing a literal control-`W` into your program.

Finally, new in Perl 5.6, Perl variable names may be alphanumeric strings that begin with control characters (or better yet, a caret). These variables must be written in the form `${^Foo}`; the braces are not optional. `${^Foo}` denotes the scalar variable whose name is 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 `^_` (control-underscore or caret-underscore). No control-character 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, control characters, 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 new special `${^_XYZ}` variables are always taken to be in package `main`, regardless of any `package` declarations presently in scope.

BUGS

Due to an unfortunate accident of Perl's implementation, `use English` imposes a considerable performance penalty on all regular expression matches in a program, regardless of whether they occur in the scope of `use English`. For that reason, saying `use English` in libraries is strongly

discouraged. See the Devel::SawAmpersand module documentation from CPAN (http://www.cpan.org/modules/by-module/Devel/) for more information. Writing `use English '-no_match_vars';` avoids the performance penalty.

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.