NAME

perlrun - how to execute the Perl interpreter

SYNOPSIS

perl [**-sTtuUWX**] [**-hv**] [**-V**[:*configvar*]] [**-cw**] [**-d**[**t**][:*debugger*]] [**-D**[*number/list*]] [**-pna**] [**-F** *pattern*] [**-l**[*octal*]] [**-0**[*octal/hexadecimal*]] [**-I***dir*] [**-m**[**-**]*module*] [**-M**[**-**]*'module...'*] [**-f**] [**-C [** ***number/list*****]**] [**-P**] [**-S**] [**-x**[*dir*]] [**-i**[*extension*]] [**-e** *'command'*] [**--**] [*programfile*] [*argument*]...

DESCRIPTION

The normal way to run a Perl program is by making it directly executable, or else by passing the name of the source file as an argument on the command line. (An interactive Perl environment is also possible--see *perldebug* for details on how to do that.) Upon startup, Perl looks for your program in one of the following places:

1. Specified line by line via **-e** switches on the command line.
2. Contained in the file specified by the first filename on the command line. (Note that systems supporting the #! notation invoke interpreters this way. See *Location of Perl*.)
3. Passed in implicitly via standard input. This works only if there are no filename arguments--to pass arguments to a STDIN-read program you must explicitly specify a "-" for the program name.

With methods 2 and 3, Perl starts parsing the input file from the beginning, unless you've specified a **-x** switch, in which case it scans for the first line starting with #! and containing the word "perl", and starts there instead. This is useful for running a program embedded in a larger message. (In this case you would indicate the end of the program using the `__END__` token.)

The #! line is always examined for switches as the line is being parsed. Thus, if you're on a machine that allows only one argument with the #! line, or worse, doesn't even recognize the #! line, you still can get consistent switch behavior regardless of how Perl was invoked, even if **-x** was used to find the beginning of the program.

Because historically some operating systems silently chopped off kernel interpretation of the #! line after 32 characters, some switches may be passed in on the command line, and some may not; you could even get a "-" without its letter, if you're not careful. You probably want to make sure that all your switches fall either before or after that 32-character boundary. Most switches don't actually care if they're processed redundantly, but getting a "-" instead of a complete switch could cause Perl to try to execute standard input instead of your program. And a partial **-I** switch could also cause odd results.

Some switches do care if they are processed twice, for instance combinations of **-l** and **-0**. Either put all the switches after the 32-character boundary (if applicable), or replace the use of **-0***digits* by `BEGIN{ $/ = "\0digits"; }`.

Parsing of the #! switches starts wherever "perl" is mentioned in the line. The sequences "-*" and "- " are specifically ignored so that you could, if you were so inclined, say

```
#!/bin/sh -- # -*- perl -*- -p
eval 'exec perl -wS $0 ${1+"$@"}'
    if $running_under_some_shell;
```

to let Perl see the **-p** switch.

A similar trick involves the **env** program, if you have it.

```
#!/usr/bin/env perl
```

The examples above use a relative path to the perl interpreter, getting whatever version is first in the

user's path. If you want a specific version of Perl, say, perl5.005_57, you should place that directly in the #! line's path.

If the #! line does not contain the word "perl", the program named after the #! is executed instead of the Perl interpreter. This is slightly bizarre, but it helps people on machines that don't do #!, because they can tell a program that their SHELL is */usr/bin/perl*, and Perl will then dispatch the program to the correct interpreter for them.

After locating your program, Perl compiles the entire program to an internal form. If there are any compilation errors, execution of the program is not attempted. (This is unlike the typical shell script, which might run part-way through before finding a syntax error.)

If the program is syntactically correct, it is executed. If the program runs off the end without hitting an exit() or die() operator, an implicit `exit(0)` is provided to indicate successful completion.

#! and quoting on non-Unix systems

Unix's #! technique can be simulated on other systems:

OS/2

Put

```
extproc perl -S -your_switches
```

as the first line in `*.cmd` file (**-S** due to a bug in cmd.exe's `extproc' handling).

MS-DOS

Create a batch file to run your program, and codify it in `ALTERNATE_SHEBANG` (see the *dosish.h* file in the source distribution for more information).

Win95/NT

The Win95/NT installation, when using the ActiveState installer for Perl, will modify the Registry to associate the *.pl* extension with the perl interpreter. If you install Perl by other means (including building from the sources), you may have to modify the Registry yourself. Note that this means you can no longer tell the difference between an executable Perl program and a Perl library file.

Macintosh

Under "Classic" MacOS, a perl program will have the appropriate Creator and Type, so that double-clicking them will invoke the MacPerl application. Under Mac OS X, clickable apps can be made from any `#!` script using Wil Sanchez' DropScript utility: http://www.wsanchez.net/software/ .

VMS

Put

```
    $ perl -mysw 'f$env("procedure")' 'p1' 'p2' 'p3' 'p4' 'p5' 'p6'
'p7' 'p8' !
    $ exit++ + ++$status != 0 and $exit = $status = undef;
```

at the top of your program, where **-mysw** are any command line switches you want to pass to Perl. You can now invoke the program directly, by saying `perl program`, or as a DCL procedure, by saying `@program` (or implicitly via *DCL$PATH* by just using the name of the program).

This incantation is a bit much to remember, but Perl will display it for you if you say `perl "-V:startperl"`.

Command-interpreters on non-Unix systems have rather different ideas on quoting than Unix shells. You'll need to learn the special characters in your command-interpreter (`*`, `\` and `"` are common) and

how to protect whitespace and these characters to run one-liners (see **-e** below).

On some systems, you may have to change single-quotes to double ones, which you must *not* do on Unix or Plan 9 systems. You might also have to change a single % to a %%.

For example:

```
# Unix
perl -e 'print "Hello world\n"'

# MS-DOS, etc.
perl -e "print \"Hello world\n\""

# Macintosh
print "Hello world\n"
 (then Run "Myscript" or Shift-Command-R)

# VMS
perl -e "print ""Hello world\n"""
```

The problem is that none of this is reliable: it depends on the command and it is entirely possible neither works. If **4DOS** were the command shell, this would probably work better:

```
perl -e "print <Ctrl-x>"Hello world\n<Ctrl-x>""
```

CMD.EXE in Windows NT slipped a lot of standard Unix functionality in when nobody was looking, but just try to find documentation for its quoting rules.

Under the Macintosh, it depends which environment you are using. The MacPerl shell, or MPW, is much like Unix shells in its support for several quoting variants, except that it makes free use of the Macintosh's non-ASCII characters as control characters.

There is no general solution to all of this. It's just a mess.

Location of Perl

It may seem obvious to say, but Perl is useful only when users can easily find it. When possible, it's good for both */usr/bin/perl* and */usr/local/bin/perl* to be symlinks to the actual binary. If that can't be done, system administrators are strongly encouraged to put (symlinks to) perl and its accompanying utilities into a directory typically found along a user's PATH, or in some other obvious and convenient place.

In this documentation, `#!/usr/bin/perl` on the first line of the program will stand in for whatever method works on your system. You are advised to use a specific path if you care about a specific version.

```
#!/usr/local/bin/perl5.00554
```

or if you just want to be running at least version, place a statement like this at the top of your program:

```
use 5.005_54;
```

Command Switches

As with all standard commands, a single-character switch may be clustered with the following switch, if any.

```
#!/usr/bin/perl -spi.orig # same as -s -p -i.orig
```

Switches include:

-0[*octal/hexadecimal*]

specifies the input record separator (`$/`) as an octal or hexadecimal number. If there are no digits, the null character is the separator. Other switches may precede or follow the digits. For example, if you have a version of **find** which can print filenames terminated by the null character, you can say this:

```
    find . -name '*.orig' -print0 | perl -n0e unlink
```

The special value 00 will cause Perl to slurp files in paragraph mode. The value 0777 will cause Perl to slurp files whole because there is no legal byte with that value.

If you want to specify any Unicode character, use the hexadecimal format: `-0xHHH...`, where the `H` are valid hexadecimal digits. (This means that you cannot use the `-x` with a directory name that consists of hexadecimal digits.)

-a

turns on autosplit mode when used with a **-n** or **-p**. An implicit split command to the @F array is done as the first thing inside the implicit while loop produced by the **-n** or **-p**.

```
    perl -ane 'print pop(@F), "\n";'
```

is equivalent to

```
    while (<>) {
 @F = split(' ');
 print pop(@F), "\n";
    }
```

An alternate delimiter may be specified using **-F**.

-C [*number/list*]

The `-C` flag controls some Unicode of the Perl Unicode features.

As of 5.8.1, the `-C` can be followed either by a number or a list of option letters. The letters, their numeric values, and effects are as follows; listing the letters is equal to summing the numbers.

```
    I     1    STDIN is assumed to be in UTF-8
    O     2    STDOUT will be in UTF-8
    E     4    STDERR will be in UTF-8
    S     7    I + O + E
    i     8    UTF-8 is the default PerlIO layer for input streams
    o    16    UTF-8 is the default PerlIO layer for output streams
    D    24    i + o
    A    32    the @ARGV elements are expected to be strings encoded
 in UTF-8
    L    64    normally the "IOEioA" are unconditional,
               the L makes them conditional on the locale
environment
               variables (the LC_ALL, LC_TYPE, and LANG, in the
order
               of decreasing precedence) -- if the variables
indicate
               UTF-8, then the selected "IOEioA" are in effect
```

For example, `-COE` and `-C6` will both turn on UTF-8-ness on both STDOUT and STDERR. Repeating letters is just redundant, not cumulative nor toggling.

The `io` options mean that any subsequent open() (or similar I/O operations) will have the

`:utf8` PerlIO layer implicitly applied to them, in other words, UTF-8 is expected from any input stream, and UTF-8 is produced to any output stream. This is just the default, with explicit layers in open() and with binmode() one can manipulate streams as usual.

`-C` on its own (not followed by any number or option list), or the empty string `""` for the `PERL_UNICODE` environment variable, has the same effect as `-CSDL`. In other words, the standard I/O handles and the default `open()` layer are UTF-8-fied **but** only if the locale environment variables indicate a UTF-8 locale. This behaviour follows the *implicit* (and problematic) UTF-8 behaviour of Perl 5.8.0.

You can use `-C0` (or `"0"` for `PERL_UNICODE`) to explicitly disable all the above Unicode features.

The read-only magic variable `${^UNICODE}` reflects the numeric value of this setting. This is variable is set during Perl startup and is thereafter read-only. If you want runtime effects, use the three-arg open() (see *"open" in perlfunc*), the two-arg binmode() (see *"binmode" in perlfunc*), and the `open` pragma (see *open*).

(In Perls earlier than 5.8.1 the `-C` switch was a Win32-only switch that enabled the use of Unicode-aware "wide system call" Win32 APIs. This feature was practically unused, however, and the command line switch was therefore "recycled".)

-c

causes Perl to check the syntax of the program and then exit without executing it. Actually, it *will* execute `BEGIN`, `CHECK`, and `use` blocks, because these are considered as occurring outside the execution of your program. `INIT` and `END` blocks, however, will be skipped.

-d

-dt

runs the program under the Perl debugger. See *perldebug*. If **t** is specified, it indicates to the debugger that threads will be used in the code being debugged.

-d:*foo[=bar,baz]*

-dt:*foo[=bar,baz]*

runs the program under the control of a debugging, profiling, or tracing module installed as Devel::foo. E.g., **-d:DProf** executes the program using the Devel::DProf profiler. As with the **-M** flag, options may be passed to the Devel::foo package where they will be received and interpreted by the Devel::foo::import routine. The comma-separated list of options must follow a = character. If **t** is specified, it indicates to the debugger that threads will be used in the code being debugged. See *perldebug*.

-D*letters*

-D*number*

sets debugging flags. To watch how it executes your program, use **-Dtls**. (This works only if debugging is compiled into your Perl.) Another nice value is **-Dx**, which lists your compiled syntax tree. And **-Dr** displays compiled regular expressions; the format of the output is explained in *perldebguts*.

As an alternative, specify a number instead of list of letters (e.g., **-D14** is equivalent to **-Dtls**):

```
        1  p  Tokenizing and parsing
        2  s  Stack snapshots (with v, displays all stacks)
        4  l  Context (loop) stack processing
        8  t  Trace execution
       16  o  Method and overloading resolution
       32  c  String/numeric conversions
       64  P  Print profiling info, preprocessor command for -P,
source file input state
      128  m  Memory allocation
```

```
      256  f  Format processing
      512  r  Regular expression parsing and execution
     1024  x  Syntax tree dump
     2048  u  Tainting checks
     4096     (Obsolete, previously used for LEAKTEST)
     8192  H  Hash dump -- usurps values()
    16384  X  Scratchpad allocation
    32768  D  Cleaning up
    65536  S  Thread synchronization
   131072  T  Tokenising
   262144  R  Include reference counts of dumped variables (eg when
using -Ds)
   524288  J  Do not s,t,P-debug (Jump over) opcodes within package
DB
  1048576  v  Verbose: use in conjunction with other flags
  8388608  q  quiet - currently only suppresses the "EXECUTING"
message
```

All these flags require **-DDEBUGGING** when you compile the Perl executable (but see *Devel::Peek*, *re* which may change this). See the *INSTALL* file in the Perl source distribution for how to do this. This flag is automatically set if you include **-g** option when `Configure` asks you about optimizer/debugger flags.

If you're just trying to get a print out of each line of Perl code as it executes, the way that `sh -x` provides for shell scripts, you can't use Perl's **-D** switch. Instead do this

```
  # If you have "env" utility
  env PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

  # Bourne shell syntax
  $ PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

  # csh syntax
  % (setenv PERLDB_OPTS "NonStop=1 AutoTrace=1 frame=2"; perl -dS
program)
```

See *perldebug* for details and variations.

-e *commandline*

may be used to enter one line of program. If **-e** is given, Perl will not look for a filename in the argument list. Multiple **-e** commands may be given to build up a multi-line script. Make sure to use semicolons where you would in a normal program.

-f

Disable executing *$Config{sitelib}/sitecustomize.pl* at startup.

Perl can be built so that it by default will try to execute *$Config{sitelib}/sitecustomize.pl* at startup. This is a hook that allows the sysadmin to customize how perl behaves. It can for instance be used to add entries to the @INC array to make perl find modules in non-standard locations.

-F*pattern*

specifies the pattern to split on if **-a** is also in effect. The pattern may be surrounded by `//`, `""`, or `''`, otherwise it will be put in single quotes. You can't use literal whitespace in the pattern.

-h

prints a summary of the options.

-i[*extension*]

specifies that files processed by the <> construct are to be edited in-place. It does this by renaming the input file, opening the output file by the original name, and selecting that output file as the default for print() statements. The extension, if supplied, is used to modify the name of the old file to make a backup copy, following these rules:

If no extension is supplied, no backup is made and the current file is overwritten.

If the extension doesn't contain a *, then it is appended to the end of the current filename as a suffix. If the extension does contain one or more * characters, then each * is replaced with the current filename. In Perl terms, you could think of this as:

```
    ($backup = $extension) =~ s/\*/$file_name/g;
```

This allows you to add a prefix to the backup file, instead of (or in addition to) a suffix:

```
    $ perl -pi'orig_*' -e 's/bar/baz/' fileA # backup to
'orig_fileA'
```

Or even to place backup copies of the original files into another directory (provided the directory already exists):

```
    $ perl -pi'old/*.orig' -e 's/bar/baz/' fileA # backup to
'old/fileA.orig'
```

These sets of one-liners are equivalent:

```
    $ perl -pi -e 's/bar/baz/' fileA  # overwrite current file
    $ perl -pi'*' -e 's/bar/baz/' fileA  # overwrite current file

    $ perl -pi'.orig' -e 's/bar/baz/' fileA # backup to 'fileA.orig'
    $ perl -pi'*.orig' -e 's/bar/baz/' fileA # backup to
'fileA.orig'
```

From the shell, saying

```
    $ perl -p -i.orig -e "s/foo/bar/; ... "
```

is the same as using the program:

```
    #!/usr/bin/perl -pi.orig
    s/foo/bar/;
```

which is equivalent to

```
    #!/usr/bin/perl
    $extension = '.orig';
    LINE: while (<>) {
 if ($ARGV ne $oldargv) {
     if ($extension !~ /\*/) {
  $backup = $ARGV . $extension;
     }
     else {
  ($backup = $extension) =~ s/\*/$ARGV/g;
     }
     rename($ARGV, $backup);
     open(ARGVOUT, ">$ARGV");
     select(ARGVOUT);
     $oldargv = $ARGV;
 }
 s/foo/bar/;
    }
```

```
    continue {
 print; # this prints to original filename
    }
    select(STDOUT);
```

except that the **-i** form doesn't need to compare $ARGV to $oldargv to know when the filename has changed. It does, however, use ARGVOUT for the selected filehandle. Note that STDOUT is restored as the default output filehandle after the loop.

As shown above, Perl creates the backup file whether or not any output is actually changed. So this is just a fancy way to copy files:

```
    $ perl -p -i'/some/file/path/*' -e 1 file1 file2 file3...
or
    $ perl -p -i'.orig' -e 1 file1 file2 file3...
```

You can use `eof` without parentheses to locate the end of each input file, in case you want to append to each file, or reset line numbering (see example in *"eof" in perlfunc*).

If, for a given file, Perl is unable to create the backup file as specified in the extension then it will skip that file and continue on with the next one (if it exists).

For a discussion of issues surrounding file permissions and **-i**, see *"Why does Perl let me delete read-only files? Why does -i clobber protected files? Isn't this a bug in Perl?" in perlfaq5*.

You cannot use **-i** to create directories or to strip extensions from files.

Perl does not expand ~ in filenames, which is good, since some folks use it for their backup files:

```
    $ perl -pi~ -e 's/foo/bar/' file1 file2 file3...
```

Note that because **-i** renames or deletes the original file before creating a new file of the same name, UNIX-style soft and hard links will not be preserved.

Finally, the **-i** switch does not impede execution when no files are given on the command line. In this case, no backup is made (the original file cannot, of course, be determined) and processing proceeds from STDIN to STDOUT as might be expected.

-I*directory*

Directories specified by **-I** are prepended to the search path for modules (`@INC`), and also tells the C preprocessor where to search for include files. The C preprocessor is invoked with **-P**; by default it searches /usr/include and /usr/lib/perl.

-l[*octnum*]

enables automatic line-ending processing. It has two separate effects. First, it automatically chomps `$/` (the input record separator) when used with **-n** or **-p**. Second, it assigns `$\` (the output record separator) to have the value of *octnum* so that any print statements will have that separator added back on. If *octnum* is omitted, sets `$\` to the current value of `$/`. For instance, to trim lines to 80 columns:

```
    perl -lpe 'substr($_, 80) = ""'
```

Note that the assignment `$\ = $/` is done when the switch is processed, so the input record separator can be different than the output record separator if the **-l** switch is followed by a **-0** switch:

```
    gnufind / -print0 | perl -ln0e 'print "found $_" if -p'
```

This sets `$\` to newline and then sets `$/` to the null character.

-m[**-**]*module*

-M[**-**]*module*

-M[**-**]*'module ...'*

-[mM][**-**]*module=arg[,arg]...*

-m*module* executes `use` *module* `();` before executing your program.

-M*module* executes `use` *module* `;` before executing your program. You can use quotes to add extra code after the module name, e.g., `'-Mmodule qw(foo bar)'`.

If the first character after the **-M** or **-m** is a dash (`-`) then the 'use' is replaced with 'no'.

A little builtin syntactic sugar means you can also say **-mmodule=foo,bar** or **-Mmodule=foo,bar** as a shortcut for `'-Mmodule qw(foo bar)'`. This avoids the need to use quotes when importing symbols. The actual code generated by **-Mmodule=foo,bar** is `use module split(/,/,q{foo,bar})`. Note that the = form removes the distinction between **-m** and **-M**.

A consequence of this is that **-MFoo=number** never does a version check (unless `Foo::import()` itself is set up to do a version check, which could happen for example if Foo inherits from Exporter.)

-n

causes Perl to assume the following loop around your program, which makes it iterate over filename arguments somewhat like **sed -n** or **awk**:

```
  LINE:
    while (<>) {
...   # your program goes here
    }
```

Note that the lines are not printed by default. See **-p** to have lines printed. If a file named by an argument cannot be opened for some reason, Perl warns you about it and moves on to the next file.

Here is an efficient way to delete all files that haven't been modified for at least a week:

```
    find . -mtime +7 -print | perl -nle unlink
```

This is faster than using the **-exec** switch of **find** because you don't have to start a process on every filename found. It does suffer from the bug of mishandling newlines in pathnames, which you can fix if you follow the example under **-0**.

`BEGIN` and `END` blocks may be used to capture control before or after the implicit program loop, just as in **awk**.

-p

causes Perl to assume the following loop around your program, which makes it iterate over filename arguments somewhat like **sed**:

```
  LINE:
    while (<>) {
...   # your program goes here
    } continue {
print or die "-p destination: $!\n";
    }
```

If a file named by an argument cannot be opened for some reason, Perl warns you about it, and moves on to the next file. Note that the lines are printed automatically. An error occurring during printing is treated as fatal. To suppress printing use the **-n** switch. A **-p** overrides a **-n** switch.

`BEGIN` and `END` blocks may be used to capture control before or after the implicit loop, just as in **awk**.

-P

NOTE: Use of -P is strongly discouraged because of its inherent problems, including poor portability.

This option causes your program to be run through the C preprocessor before compilation by Perl. Because both comments and **cpp** directives begin with the # character, you should avoid starting comments with any words recognized by the C preprocessor such as `"if"`, `"else"`, or `"define"`.

If you're considering using `-P`, you might also want to look at the Filter::cpp module from CPAN.

The problems of -P include, but are not limited to:

- The `#!` line is stripped, so any switches there don't apply.
- A `-P` on a `#!` line doesn't work.
- **All** lines that begin with (whitespace and) a # but do not look like cpp commands, are stripped, including anything inside Perl strings, regular expressions, and here-docs .
- In some platforms the C preprocessor knows too much: it knows about the C++ -style until-end-of-line comments starting with `"//"`. This will cause problems with common Perl constructs like

  ```
  s/foo//;
  ```

  because after -P this will became illegal code

  ```
  s/foo
  ```

  The workaround is to use some other quoting separator than `"/"`, like for example `"!"`:

  ```
  s!foo!;
  ```

- It requires not only a working C preprocessor but also a working *sed*. If not on UNIX, you are probably out of luck on this.
- Script line numbers are not preserved.
- The `-x` does not work with `-P`.

-s

enables rudimentary switch parsing for switches on the command line after the program name but before any filename arguments (or before an argument of **--**). Any switch found there is removed from @ARGV and sets the corresponding variable in the Perl program. The following program prints "1" if the program is invoked with a **-xyz** switch, and "abc" if it is invoked with **-xyz=abc**.

```
#!/usr/bin/perl -s
if ($xyz) { print "$xyz\n" }
```

Do note that a switch like **--help** creates the variable ${-help}, which is not compliant with `strict refs`. Also, when using this option on a script with warnings enabled you may get a lot of spurious "used only once" warnings.

-S

makes Perl use the PATH environment variable to search for the program (unless the name of the program contains directory separators).

On some platforms, this also makes Perl append suffixes to the filename while searching for

it. For example, on Win32 platforms, the ".bat" and ".cmd" suffixes are appended if a lookup for the original name fails, and if the name does not already end in one of those suffixes. If your Perl was compiled with DEBUGGING turned on, using the -Dp switch to Perl shows how the search progresses.

Typically this is used to emulate #! startup on platforms that don't support #!. Its also convenient when debugging a script that uses #!, and is thus normally found by the shell's $PATH search mechanism.

This example works on many platforms that have a shell compatible with Bourne shell:

```
#!/usr/bin/perl
eval 'exec /usr/bin/perl -wS $0 ${1+"$@"}'
 if $running_under_some_shell;
```

The system ignores the first line and feeds the program to */bin/sh*, which proceeds to try to execute the Perl program as a shell script. The shell executes the second line as a normal shell command, and thus starts up the Perl interpreter. On some systems $0 doesn't always contain the full pathname, so the **-S** tells Perl to search for the program if necessary. After Perl locates the program, it parses the lines and ignores them because the variable $running_under_some_shell is never true. If the program will be interpreted by csh, you will need to replace `${1+"$@"}` with `$*`, even though that doesn't understand embedded spaces (and such) in the argument list. To start up sh rather than csh, some systems may have to replace the #! line with a line containing just a colon, which will be politely ignored by Perl. Other systems can't control that, and need a totally devious construct that will work under any of **csh**, **sh**, or Perl, such as the following:

```
eval '(exit $?0)' && eval 'exec perl -wS $0 ${1+"$@"}'
& eval 'exec /usr/bin/perl -wS $0 $argv:q'
 if $running_under_some_shell;
```

If the filename supplied contains directory separators (i.e., is an absolute or relative pathname), and if that file is not found, platforms that append file extensions will do so and try to look for the file with those extensions added, one by one.

On DOS-like platforms, if the program does not contain directory separators, it will first be searched for in the current directory before being searched for on the PATH. On Unix platforms, the program will be searched for strictly on the PATH.

-t

Like **-T**, but taint checks will issue warnings rather than fatal errors. These warnings can be controlled normally with `no warnings qw(taint)`.

NOTE: this is not a substitute for -T. This is meant only to be used as a temporary development aid while securing legacy code: for real production code and for new secure code written from scratch always use the real **-T**.

-T

forces "taint" checks to be turned on so you can test them. Ordinarily these checks are done only when running setuid or setgid. It's a good idea to turn them on explicitly for programs that run on behalf of someone else whom you might not necessarily trust, such as CGI programs or any internet servers you might write in Perl. See *perlsec* for details. For security reasons, this option must be seen by Perl quite early; usually this means it must appear early on the command line or in the #! line for systems which support that construct.

-u

This obsolete switch causes Perl to dump core after compiling your program. You can then in theory take this core dump and turn it into an executable file by using the **undump** program (not supplied). This speeds startup at the expense of some disk space (which you can minimize by stripping the executable). (Still, a "hello world" executable comes out to

about 200K on my machine.) If you want to execute a portion of your program before dumping, use the dump() operator instead. Note: availability of **undump** is platform specific and may not be available for a specific port of Perl.

This switch has been superseded in favor of the new Perl code generator backends to the compiler. See *B* and *B::Bytecode* for details.

-U

allows Perl to do unsafe operations. Currently the only "unsafe" operations are attempting to unlink directories while running as superuser, and running setuid programs with fatal taint checks turned into warnings. Note that the **-w** switch (or the `$^W` variable) must be used along with this option to actually *generate* the taint-check warnings.

-v

prints the version and patchlevel of your perl executable.

-V

prints summary of the major perl configuration values and the current values of @INC.

-V:*configvar*

Prints to STDOUT the value of the named configuration variable(s), with multiples when your configvar argument looks like a regex (has non-letters). For example:

```
    $ perl -V:libc
 libc='/lib/libc-2.2.4.so';
    $ perl -V:lib.
 libs='-lnsl -lgdbm -ldb -ldl -lm -lcrypt -lutil -lc';
 libc='/lib/libc-2.2.4.so';
    $ perl -V:lib.*
 libpth='/usr/local/lib /lib /usr/lib';
 libs='-lnsl -lgdbm -ldb -ldl -lm -lcrypt -lutil -lc';
 lib_ext='.a';
 libc='/lib/libc-2.2.4.so';
 libperl='libperl.a';
 ....
```

Additionally, extra colons can be used to control formatting. A trailing colon suppresses the linefeed and terminator ';', allowing you to embed queries into shell commands. (mnemonic: PATH separator ':'.)

```
    $ echo "compression-vars: " `perl -V:z.*: ` " are here !"
    compression-vars:  zcat='' zip='zip'  are here !
```

A leading colon removes the 'name=' part of the response, this allows you to map to the name you need. (mnemonic: empty label)

```
    $ echo "goodvfork="`./perl -Ilib -V::usevfork`
    goodvfork=false;
```

Leading and trailing colons can be used together if you need positional parameter values without the names. Note that in the case below, the PERL_API params are returned in alphabetical order.

```
    $ echo building_on `perl -V::osname: -V::PERL_API_.*:` now
    building_on 'linux' '5' '1' '9' now
```

-w

prints warnings about dubious constructs, such as variable names that are mentioned only once and scalar variables that are used before being set, redefined subroutines, references

to undefined filehandles or filehandles opened read-only that you are attempting to write on, values used as a number that don't look like numbers, using an array as though it were a scalar, if your subroutines recurse more than 100 deep, and innumerable other things.

This switch really just enables the internal `$^W` variable. You can disable or promote into fatal errors specific warnings using `__WARN__` hooks, as described in *perlvar* and *"warn" in perlfunc*. See also *perldiag* and *perltrap*. A new, fine-grained warning facility is also available if you want to manipulate entire classes of warnings; see *warnings* or *perllexwarn*.

-W

Enables all warnings regardless of `no warnings` or `$^W`. See *perllexwarn*.

-X

Disables all warnings regardless of `use warnings` or `$^W`. See *perllexwarn*.

-x

-x *directory*

tells Perl that the program is embedded in a larger chunk of unrelated ASCII text, such as in a mail message. Leading garbage will be discarded until the first line that starts with #! and contains the string "perl". Any meaningful switches on that line will be applied. If a directory name is specified, Perl will switch to that directory before running the program. The **-x** switch controls only the disposal of leading garbage. The program must be terminated with `__END__` if there is trailing garbage to be ignored (the program can process any or all of the trailing garbage via the DATA filehandle if desired).

ENVIRONMENT

HOME

Used if chdir has no argument.

LOGDIR

Used if chdir has no argument and HOME is not set.

PATH

Used in executing subprocesses, and in finding the program if **-S** is used.

PERL5LIB

A list of directories in which to look for Perl library files before looking in the standard library and the current directory. Any architecture-specific directories under the specified locations are automatically included if they exist. If PERL5LIB is not defined, PERLLIB is used. Directories are separated (like in PATH) by a colon on unixish platforms and by a semicolon on Windows (the proper path separator being given by the command `perl -V:path_sep`).

When running taint checks (either because the program was running setuid or setgid, or the **-T** switch was used), neither variable is used. The program should instead say:

```
use lib "/my/directory";
```

PERL5OPT

Command-line options (switches). Switches in this variable are taken as if they were on every Perl command line. Only the **-[DIMUdmtw]** switches are allowed. When running taint checks (because the program was running setuid or setgid, or the **-T** switch was used), this variable is ignored. If PERL5OPT begins with **-T**, tainting will be enabled, and any subsequent options ignored.

PERLIO

A space (or colon) separated list of PerlIO layers. If perl is built to use PerlIO system for IO (the default) these layers effect perl's IO.

It is conventional to start layer names with a colon e.g. `:perlio` to emphasise their similarity to variable "attributes". But the code that parses layer specification strings (which is also used to decode the PERLIO environment variable) treats the colon as a separator.

An unset or empty PERLIO is equivalent to `:stdio`.

The list becomes the default for *all* perl's IO. Consequently only built-in layers can appear in this list, as external layers (such as :encoding()) need IO in order to load them!. See *open pragma* for how to add external encodings as defaults.

The layers that it makes sense to include in the PERLIO environment variable are briefly summarised below. For more details see *PerlIO*.

:bytes

A pseudolayer that turns *off* the `:utf8` flag for the layer below. Unlikely to be useful on its own in the global PERLIO environment variable. You perhaps were thinking of `:crlf:bytes` or `:perlio:bytes`.

:crlf

A layer which does CRLF to "\n" translation distinguishing "text" and "binary" files in the manner of MS-DOS and similar operating systems. (It currently does *not* mimic MS-DOS as far as treating of Control-Z as being an end-of-file marker.)

:mmap

A layer which implements "reading" of files by using `mmap()` to make (whole) file appear in the process's address space, and then using that as PerlIO's "buffer".

:perlio

This is a re-implementation of "stdio-like" buffering written as a PerlIO "layer". As such it will call whatever layer is below it for its operations (typically `:unix`).

:pop

An experimental pseudolayer that removes the topmost layer. Use with the same care as is reserved for nitroglycerin.

:raw

A pseudolayer that manipulates other layers. Applying the `:raw` layer is equivalent to calling `binmode($fh)`. It makes the stream pass each byte as-is without any translation. In particular CRLF translation, and/or :utf8 intuited from locale are disabled.

Unlike in the earlier versions of Perl `:raw` is *not* just the inverse of `:crlf` - other layers which would affect the binary nature of the stream are also removed or disabled.

:stdio

This layer provides PerlIO interface by wrapping system's ANSI C "stdio" library calls. The layer provides both buffering and IO. Note that `:stdio` layer does *not* do CRLF translation even if that is platforms normal behaviour. You will need a `:crlf` layer above it to do that.

:unix

Low level layer which calls `read`, `write` and `lseek` etc.

:utf8

A pseudolayer that turns on a flag on the layer below to tell perl that output should be in utf8 and that input should be regarded as already in utf8 form. May be useful in PERLIO environment variable to make UTF-8 the default. (To turn off that behaviour use `:bytes` layer.)

:win32

On Win32 platforms this *experimental* layer uses native "handle" IO rather than unix-like numeric file descriptor layer. Known to be buggy in this release.

On all platforms the default set of layers should give acceptable results.

For UNIX platforms that will equivalent of "unix perlio" or "stdio". Configure is setup to prefer "stdio" implementation if system's library provides for fast access to the buffer, otherwise it uses the "unix perlio" implementation.

On Win32 the default in this release is "unix crlf". Win32's "stdio" has a number of bugs/mis-features for perl IO which are somewhat C compiler vendor/version dependent. Using our own `crlf` layer as the buffer avoids those issues and makes things more uniform. The `crlf` layer provides CRLF to/from "\n" conversion as well as buffering.

This release uses `unix` as the bottom layer on Win32 and so still uses C compiler's numeric file descriptor routines. There is an experimental native `win32` layer which is expected to be enhanced and should eventually be the default under Win32.

PERLIO_DEBUG

If set to the name of a file or device then certain operations of PerlIO sub-system will be logged to that file (opened as append). Typical uses are UNIX:

```
PERLIO_DEBUG=/dev/tty perl script ...
```

and Win32 approximate equivalent:

```
set PERLIO_DEBUG=CON
perl script ...
```

This functionality is disabled for setuid scripts and for scripts run with **-T**.

PERLLIB

A list of directories in which to look for Perl library files before looking in the standard library and the current directory. If PERL5LIB is defined, PERLLIB is not used.

PERL5DB

The command used to load the debugger code. The default is:

```
BEGIN { require 'perl5db.pl' }
```

PERL5DB_THREADED

If set to a true value, indicates to the debugger that the code being debugged uses threads.

PERL5SHELL (specific to the Win32 port)

May be set to an alternative shell that perl must use internally for executing "backtick" commands or system(). Default is `cmd.exe /x/d/c` on WindowsNT and `command.com /c` on Windows95. The value is considered to be space-separated. Precede any character that needs to be protected (like a space or backslash) with a backslash.

Note that Perl doesn't use COMSPEC for this purpose because COMSPEC has a high degree of variability among users, leading to portability concerns. Besides, perl can use a shell that may not be fit for interactive use, and setting COMSPEC to such a shell may interfere with the proper functioning of other programs (which usually look in COMSPEC to find a shell fit for interactive use).

PERL_ALLOW_NON_IFS_LSP (specific to the Win32 port)

Set to 1 to allow the use of non-IFS compatible LSP's. Perl normally searches for an IFS-compatible LSP because this is required for its emulation of Windows sockets as real filehandles. However, this may cause problems if you have a firewall such as McAfee Guardian which requires all applications to use its LSP which is not IFS-compatible, because clearly Perl will normally avoid using such an LSP. Setting this environment variable to 1 means that Perl will simply use the first suitable LSP enumerated in the catalog, which keeps McAfee Guardian happy (and in that particular case Perl still works too because McAfee Guardian's LSP actually plays some other games which allow applications requiring IFS compatibility to work).

PERL_DEBUG_MSTATS

Relevant only if perl is compiled with the malloc included with the perl distribution (that is, if `perl -V:d_mymalloc` is 'define'). If set, this causes memory statistics to be dumped after execution. If set to an integer greater than one, also causes memory statistics to be dumped after compilation.

PERL_DESTRUCT_LEVEL

Relevant only if your perl executable was built with **-DDEBUGGING**, this controls the behavior of global destruction of objects and other references. See *"PERL_DESTRUCT_LEVEL" in perlhack* for more information.

PERL_DL_NONLAZY

Set to one to have perl resolve **all** undefined symbols when it loads a dynamic library. The default behaviour is to resolve symbols when they are used. Setting this variable is useful during testing of extensions as it ensures that you get an error on misspelled function names even if the test suite doesn't call it.

PERL_ENCODING

If using the `encoding` pragma without an explicit encoding name, the PERL_ENCODING environment variable is consulted for an encoding name.

PERL_HASH_SEED

(Since Perl 5.8.1.) Used to randomise Perl's internal hash function. To emulate the pre-5.8.1 behaviour, set to an integer (zero means exactly the same order as 5.8.0). "Pre-5.8.1" means, among other things, that hash keys will be ordered the same between different runs of Perl.

The default behaviour is to randomise unless the PERL_HASH_SEED is set. If Perl has been compiled with `-DUSE_HASH_SEED_EXPLICIT`, the default behaviour is **not** to randomise unless the PERL_HASH_SEED is set.

If PERL_HASH_SEED is unset or set to a non-numeric string, Perl uses the

pseudorandom seed supplied by the operating system and libraries. This means that each different run of Perl will have a different ordering of the results of keys(), values(), and each().

Please note that the hash seed is sensitive information. Hashes are randomized to protect against local and remote attacks against Perl code. By manually setting a seed this protection may be partially or completely lost.

See *"Algorithmic Complexity Attacks" in perlsec* and *PERL_HASH_SEED_DEBUG* for more information.

PERL_HASH_SEED_DEBUG

(Since Perl 5.8.1.) Set to one to display (to STDERR) the value of the hash seed at the beginning of execution. This, combined with *PERL_HASH_SEED* is intended to aid in debugging nondeterministic behavior caused by hash randomization.

Note that the hash seed is sensitive information: by knowing it one can craft a denial-of-service attack against Perl code, even remotely, see *"Algorithmic Complexity Attacks" in perlsec* for more information. **Do not disclose the hash seed** to people who don't need to know it. See also hash_seed() of *Hash::Util*.

PERL_ROOT (specific to the VMS port)

A translation concealed rooted logical name that contains perl and the logical device for the @INC path on VMS only. Other logical names that affect perl on VMS include PERLSHR, PERL_ENV_TABLES, and SYS$TIMEZONE_DIFFERENTIAL but are optional and discussed further in *perlvms* and in *README.vms* in the Perl source distribution.

PERL_SIGNALS

In Perls 5.8.1 and later. If set to `unsafe` the pre-Perl-5.8.0 signals behaviour (immediate but unsafe) is restored. If set to `safe` the safe (or deferred) signals are used. See *"Deferred Signals (Safe Signals)" in perlipc*.

PERL_UNICODE

Equivalent to the **-C** command-line switch. Note that this is not a boolean variable-- setting this to `"1"` is not the right way to "enable Unicode" (whatever that would mean). You can use `"0"` to "disable Unicode", though (or alternatively unset PERL_UNICODE in your shell before starting Perl). See the description of the `-C` switch for more information.

SYS$LOGIN (specific to the VMS port)

Used if chdir has no argument and HOME and LOGDIR are not set.

Perl also has environment variables that control how Perl handles data specific to particular natural languages. See *perllocale*.

Apart from these, Perl uses no other environment variables, except to make them available to the program being executed, and to child processes. However, programs running setuid would do well to execute the following lines before doing anything else, just to keep people honest:

```
$ENV{PATH}  = '/bin:/usr/bin';    # or whatever you need
$ENV{SHELL} = '/bin/sh' if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```