



Full credit is given to the above companies including the Operating System (OS) that this PDF file was generated!

Rocky Enterprise Linux 9.2 Manual Pages on command 'perldeprecation.1'

\$ man perldeprecation.1

PERLDEPRECATION(1) Perl Programmers Reference Guide PERLDEPRECATION(1)

NAME

perldeprecation - list Perl deprecations

DESCRIPTION

The purpose of this document is to document what has been deprecated in Perl, and by which version the deprecated feature will disappear, or, for already removed features, when it was removed.

This document will try to discuss what alternatives for the deprecated features are available.

The deprecated features will be grouped by the version of Perl in which they will be removed.

Perl 5.34

There are no deprecations or fatalizations scheduled for Perl 5.34.

Perl 5.32

Constants from lexical variables potentially modified elsewhere

You wrote something like

```
my $var;  
$sub = sub () { $var };
```

but `$var` is referenced elsewhere and could be modified after the "sub" expression is evaluated. Either it is explicitly modified elsewhere ("`$var = 3`") or it is passed to a subroutine or to an operator like "printf" or "map", which may or may not modify the variable.

Traditionally, Perl has captured the value of the variable at that point and turned the subroutine into a constant eligible for inlining. In those cases where the variable can be modified elsewhere, this breaks the behavior of closures, in which the subroutine captures the variable itself, rather than its value, so future changes to the variable are reflected in the subroutine's return value.

If you intended for the subroutine to be eligible for inlining, then make sure the variable is not referenced elsewhere, possibly by copying it:

```
my $var2 = $var;  
$sub = sub () { $var2 };
```

If you do want this subroutine to be a closure that reflects future changes to the variable that it closes over, add an explicit "return":

```
my $var;  
$sub = sub () { return $var };
```

This usage was deprecated and as of Perl 5.32 is no longer allowed.

Use of strings with code points over 0xFF as arguments to "vec"

"vec" views its string argument as a sequence of bits. A string containing a code point over 0xFF is nonsensical. This usage is deprecated in Perl 5.28, and was removed in Perl

5.32.

Use of code points over 0xFF in string bitwise operators

The string bitwise operators, "&", "|", "^", and "~", treat their operands as strings of bytes. As such, values above 0xFF are nonsensical. Some instances of these have been deprecated since Perl 5.24, and were made fatal in 5.28, but it turns out that in cases where the wide characters did not affect the end result, no deprecation notice was raised, and so remain legal. Now, all occurrences either are fatal or raise a deprecation warning, so that the remaining legal occurrences became fatal in 5.32.

An example of this is

```
"" & "\x{100}"
```

The wide character is not used in the "&" operation because the left operand is shorter. This now throws an exception.

hostname() doesn't accept any arguments

The function "hostname()" in the Sys::Hostname module has always been documented to be called with no arguments. Historically it has not enforced this, and has actually accepted and ignored any arguments. As a result, some users have got the mistaken impression that an argument does something useful. To avoid these bugs, the function is being made strict. Passing arguments was deprecated in Perl 5.28 and became fatal in Perl 5.32.

Unescaped left braces in regular expressions

The simple rule to remember, if you want to match a literal "{" character (U+007B "LEFT CURLY BRACKET") in a regular expression pattern, is to escape each literal instance of it in some way. Generally easiest is to precede it with a backslash, like "\{" or enclose it in square brackets ("[{]"). If the pattern delimiters are also braces, any matching right

brace ("}") should also be escaped to avoid confusing the parser, for example,

```
qr{abc\{def\}ghi}
```

Forcing literal "{" characters to be escaped will enable the Perl language to be extended in various ways in future releases. To avoid needlessly breaking existing code, the restriction is not enforced in contexts where there are unlikely to ever be extensions that could conflict with the use there of "{" as a literal. A non-deprecation warning that the left brace is being taken literally is raised in contexts where there could be confusion about it.

Literal uses of "{" were deprecated in Perl 5.20, and some uses of it started to give deprecation warnings since. These cases were made fatal in Perl 5.26. Due to an oversight, not all cases of a use of a literal "{" got a deprecation warning. Some cases started warning in Perl 5.26, and were made fatal in Perl 5.30. Other cases started in Perl 5.28, and were made fatal in 5.32.

In XS code, use of various macros dealing with UTF-8.

The macros below now require an extra parameter than in versions prior to Perl 5.32. The final parameter in each one is a pointer into the string supplied by the first parameter beyond which the input will not be read. This prevents potential reading beyond the end of the buffer. "isALPHANUMERIC_utf8", "isASCII_utf8", "isBLANK_utf8", "isCNTRL_utf8", "isDIGIT_utf8", "isIDFIRST_utf8", "isPSXSPC_utf8", "isSPACE_utf8", "isVERTWS_utf8", "isWORDCHAR_utf8", "isXDIGIT_utf8", "isALPHANUMERIC_LC_utf8", "isALPHA_LC_utf8", "isASCII_LC_utf8", "isBLANK_LC_utf8", "isCNTRL_LC_utf8", "isDIGIT_LC_utf8", "isGRAPH_LC_utf8", "isIDCONT_LC_utf8", "isIDFIRST_LC_utf8", "isLOWER_LC_utf8", "isPRINT_LC_utf8", "isPSXSPC_LC_utf8", "isPUNCT_LC_utf8", "isSPACE_LC_utf8", "isUPPER_LC_utf8", "isWORDCHAR_LC_utf8", "isXDIGIT_LC_utf8", "toFOLD_utf8", "toLOWER_utf8", "toTITLE_utf8", and "toUPPER_utf8".

Since Perl 5.26, this functionality with the extra parameter has been available by using a corresponding macro to each one of these, and whose name is formed by appending "_safe" to

the base name. There is no change to the functionality of those. For example, "isDIGIT_utf8_safe" corresponds to "isDIGIT_utf8", and both now behave identically. All are documented in "Character case changing" in perlapi and "Character classification" in perlapi.

This change was originally scheduled for 5.30, but was delayed until 5.32.

"File::Glob::glob()" was removed

"File::Glob" has a function called "glob", which just calls "bsd_glob".

"File::Glob::glob()" was deprecated in Perl 5.8. A deprecation message was issued from Perl 5.26 onwards, and the function has now disappeared in Perl 5.30.

Code using "File::Glob::glob()" should call "File::Glob::bsd_glob()" instead.

Perl 5.30

`$*` is no longer supported

Before Perl 5.10, setting `$*` to a true value globally enabled multi-line matching within a string. This relic from the past lost its special meaning in 5.10. Use of this variable will be a fatal error in Perl 5.30, freeing the variable up for a future special meaning.

To enable multiline matching one should use the `/m` regexp modifier (possibly in combination with `/s`). This can be set on a per match bases, or can be enabled per lexical scope (including a whole file) with `use re '/m'`.

`$#` is no longer supported

This variable used to have a special meaning -- it could be used to control how numbers were formatted when printed. This seldom used functionality was removed in Perl 5.10. In order to free up the variable for a future special meaning, its use will be a fatal error in Perl 5.30.

To specify how numbers are formatted when printed, one is advised to use "printf" or "sprintf" instead.

Assigning non-zero to \$[is fatal

This variable (and the corresponding "array_base" feature and arybase module) allowed changing the base for array and string indexing operations.

Setting this to a non-zero value has been deprecated since Perl 5.12 and throws a fatal error as of Perl 5.30.

"File::Glob::glob()" will disappear

"File::Glob" has a function called "glob", which just calls "bsd_glob". However, its prototype is different from the prototype of "CORE::glob", and hence, "File::Glob::glob" should not be used.

"File::Glob::glob()" was deprecated in Perl 5.8. A deprecation message was issued from Perl 5.26 onwards, and the function will disappear in Perl 5.30.

Code using "File::Glob::glob()" should call "File::Glob::bsd_glob()" instead.

Unescaped left braces in regular expressions (for 5.30)

See "Unescaped left braces in regular expressions" above.

Unqualified "dump()"

Use of "dump()" instead of "CORE::dump()" was deprecated in Perl 5.8, and an unqualified "dump()" will no longer be available in Perl 5.30.

See "dump" in perlfunc.

Using my() in false conditional.

There has been a long-standing bug in Perl that causes a lexical variable not to be cleared at scope exit when its declaration includes a false conditional. Some people have exploited this bug to achieve a kind of static variable. To allow us to fix this bug, people should not be relying on this behavior.

Instead, it's recommended one uses "state" variables to achieve the same effect:

```
use 5.10.0;
sub count {state $counter; return ++ $counter}
say count (); # Prints 1
say count (); # Prints 2
```

"state" variables were introduced in Perl 5.10.

Alternatively, you can achieve a similar static effect by declaring the variable in a separate block outside the function, e.g.,

```
sub f { my $x if 0; return $x++ }
```

becomes

```
{ my $x; sub f { return $x++ } }
```

The use of "my()" in a false conditional has been deprecated in Perl 5.10, and became a fatal error in Perl 5.30.

Reading/writing bytes from/to :utf8 handles.

The sysread(), recv(), syswrite() and send() operators are deprecated on handles that have the ":utf8" layer, either explicitly, or implicitly, eg., with the ":encoding(UTF-16LE)"

layer.

Both `sysread()` and `recv()` currently use only the `":utf8"` flag for the stream, ignoring the actual layers. Since `sysread()` and `recv()` do no UTF-8 validation they can end up creating invalidly encoded scalars.

Similarly, `syswrite()` and `send()` use only the `":utf8"` flag, otherwise ignoring any layers. If the flag is set, both write the value UTF-8 encoded, even if the layer is some different encoding, such as the example above.

Ideally, all of these operators would completely ignore the `":utf8"` state, working only with bytes, but this would result in silently breaking existing code. To avoid this a future version of perl will throw an exception when any of `sysread()`, `recv()`, `syswrite()` or `send()` are called on handle with the `":utf8"` layer.

In Perl 5.30, it will no longer be possible to use `sysread()`, `recv()`, `syswrite()` or `send()` to read or send bytes from/to `:utf8` handles.

Use of unassigned code point or non-standalone grapheme for a delimiter.

A grapheme is what appears to a native-speaker of a language to be a character. In Unicode (and hence Perl) a grapheme may actually be several adjacent characters that together form a complete grapheme. For example, there can be a base character, like "R" and an accent, like a circumflex "^", that appear to be a single character when displayed, with the circumflex hovering over the "R".

As of Perl 5.30, use of delimiters which are non-standalone graphemes is fatal, in order to move the language to be able to accept multi-character graphemes as delimiters.

Also, as of Perl 5.30, delimiters which are unassigned code points but that may someday become assigned are prohibited. Otherwise, code that works today would fail to compile if the currently unassigned delimiter ends up being something that isn't a stand-alone grapheme. Because Unicode is never going to assign non-character code points, nor code

points that are above the legal Unicode maximum, those can be delimiters.

Perl 5.28

Attributes `":locked"` and `":unique"`

The attributes `":locked"` (on code references) and `":unique"` (on array, hash and scalar references) have had no effect since Perl 5.005 and Perl 5.8.8 respectively. Their use has been deprecated since.

As of Perl 5.28, these attributes are syntax errors. Since the attributes do not do anything, removing them from your code fixes the syntax error; and removing them will not influence the behaviour of your code.

Bare here-document terminators

Perl has allowed you to use a bare here-document terminator to have the here-document end at the first empty line. This practise was deprecated in Perl 5.000; as of Perl 5.28, using a bare here-document terminator throws a fatal error.

You are encouraged to use the explicitly quoted form if you wish to use an empty line as the terminator of the here-document:

```
print <<"";
```

Print this line.

```
# Previous blank line ends the here-document.
```

Setting `$/` to a reference to a non-positive integer

You assigned a reference to a scalar to `$/` where the referenced item is not a positive integer. In older perls this appeared to work the same as setting it to `"undef"` but was in fact internally different, less efficient and with very bad luck could have resulted in your file being split by a stringified form of the reference.

In Perl 5.20.0 this was changed so that it would be exactly the same as setting `$/` to `undef`, with the exception that this warning would be thrown.

As of Perl 5.28, setting `$/` to a reference of a non-positive integer throws a fatal error.

You are recommended to change your code to set `$/` to `"undef"` explicitly if you wish to slurp the file.

Limit on the value of Unicode code points.

Unicode only allows code points up to `0x10FFFF`, but Perl allows much larger ones. Up till Perl 5.28, it was allowed to use code points exceeding the maximum value of an integer (`"IV_MAX"`). However, that did break the perl interpreter in some constructs, including causing it to hang in a few cases. The known problem areas were in `"tr///"`, regular expression pattern matching using quantifiers, as quote delimiters in `"qX...X"` (where X is the `"chr()"` of a large code point), and as the upper limits in loops.

The use of out of range code points was deprecated in Perl 5.24; as of Perl 5.28 using a code point exceeding `"IV_MAX"` throws a fatal error.

If your code is to run on various platforms, keep in mind that the upper limit depends on the platform. It is much larger on 64-bit word sizes than 32-bit ones. For 32-bit integers, `"IV_MAX"` equals `0x7FFFFFFF`, for 64-bit integers, `"IV_MAX"` equals `0x7FFFFFFFFFFFFFFF`.

Use of comma-less variable list in formats.

It was allowed to use a list of variables in a format, without separating them with commas. This usage has been deprecated for a long time, and as of Perl 5.28, this throws a fatal error.

Use of `"\N{"`

Use of "\N{}" with nothing between the braces was deprecated in Perl 5.24, and throws a fatal error as of Perl 5.28.

Since such a construct is equivalent to using an empty string, you are recommended to remove such "\N{}" constructs.

Using the same symbol to open a filehandle and a dirhandle

It used to be legal to use "open()" to associate both a filehandle and a dirhandle to the same symbol (glob or scalar). This idiom is likely to be confusing, and it was deprecated in Perl 5.10.

Using the same symbol to "open()" a filehandle and a dirhandle throws a fatal error as of Perl 5.28.

You should be using two different symbols instead.

`${^ENCODING}` is no longer supported.

The special variable "`${^ENCODING}`" was used to implement the "encoding" pragma. Setting this variable to anything other than "undef" was deprecated in Perl 5.22. Full deprecation of the variable happened in Perl 5.25.3.

Setting this variable to anything other than an undefined value throws a fatal error as of Perl 5.28.

`"B::OP::terse"`

This method, which just calls `"B::Concise::b_terse"`, has been deprecated, and disappeared in Perl 5.28. Please use `"B::Concise"` instead.

Use of inherited AUTOLOAD for non-method `%s::%s()` is no longer allowed

As an (ahem) accidental feature, "AUTOLOAD" subroutines were looked up as methods (using the @ISA hierarchy) even when the subroutines to be autoloaded were called as plain functions (e.g. "Foo::bar()"), not as methods (e.g. "Foo->bar()" or "\$obj->bar()").

This bug was deprecated in Perl 5.004, has been rectified in Perl 5.28 by using method lookup only for methods' "AUTOLOAD"s.

The simple rule is: Inheritance will not work when autoloading non-methods. The simple fix for old code is: In any module that used to depend on inheriting "AUTOLOAD" for non-methods from a base class named "BaseClass", execute `**AUTOLOAD = \&BaseClass::AUTOLOAD` during startup.

In code that currently says `use AutoLoader; @ISA = qw(AutoLoader);` you should remove AutoLoader from @ISA and change `use AutoLoader;` to `use AutoLoader 'AUTOLOAD';`.

Use of code points over 0xFF in string bitwise operators

The string bitwise operators, "&", "|", "^", and "~", treat their operands as strings of bytes. As such, values above 0xFF are nonsensical. Using such code points with these operators was deprecated in Perl 5.24, and is fatal as of Perl 5.28.

In XS code, use of `to_utf8_case()`

This function has been removed as of Perl 5.28; instead convert to call the appropriate one of: `toFOLD_utf8_safe`, `toLOWER_utf8_safe`, `toTITLE_utf8_safe`, or `toUPPER_utf8_safe`.

Perl 5.26

`--libpods` in `Pod::Html`

Since Perl 5.18, the option `--libpods` has been deprecated, and using this option did not do anything other than producing a warning.

The "--libpods" option is no longer recognized as of Perl 5.26.

The utilities "c2ph" and "pstruct"

These old, perl3-era utilities have been deprecated in favour of "h2xs" for a long time. As of Perl 5.26, they have been removed.

Trapping "\$SIG {__DIE__}" other than during program exit.

The \$SIG{__DIE__} hook is called even inside an "eval()". It was never intended to happen this way, but an implementation glitch made this possible. This used to be deprecated, as it allowed strange action at a distance like rewriting a pending exception in \$@. Plans to rectify this have been scrapped, as users found that rewriting a pending exception is actually a useful feature, and not a bug.

Perl never issued a deprecation warning for this; the deprecation was by documentation policy only. But this deprecation has been lifted as of Perl 5.26.

Malformed UTF-8 string in "%s"

This message indicates a bug either in the Perl core or in XS code. Such code was trying to find out if a character, allegedly stored internally encoded as UTF-8, was of a given type, such as being punctuation or a digit. But the character was not encoded in legal UTF-8. The %s is replaced by a string that can be used by knowledgeable people to determine what the type being checked against was.

Passing malformed strings was deprecated in Perl 5.18, and became fatal in Perl 5.26.

Perl 5.24

Use of *glob{FILEHANDLE}

The use of *glob{FILEHANDLE} was deprecated in Perl 5.8. The intention was to use

*glob{IO} instead, for which *glob{FILEHANDLE} is an alias.

However, this feature was undeprecated in Perl 5.24.

Calling POSIX::%s() is deprecated

The following functions in the "POSIX" module are no longer available: "isalnum", "isalpha", "iscntrl", "isdigit", "isgraph", "islower", "isprint", "ispunct", "isspace", "isupper", and "isxdigit". The functions are buggy and don't work on UTF-8 encoded strings. See their entries in POSIX for more information.

The functions were deprecated in Perl 5.20, and removed in Perl 5.24.

Perl 5.16

Use of %s on a handle without * is deprecated

It used to be possible to use "tie", "tied" or "untie" on a scalar while the scalar holds a typeglob. This caused its filehandle to be tied. It left no way to tie the scalar itself when it held a typeglob, and no way to untie a scalar that had had a typeglob assigned to it.

This was deprecated in Perl 5.14, and the bug was fixed in Perl 5.16.

So now "tie \$scalar" will always tie the scalar, not the handle it holds. To tie the handle, use "tie *\$scalar" (with an explicit asterisk). The same applies to "tied *\$scalar" and "untie *\$scalar".

SEE ALSO

warnings, diagnostics.