NAME

perl56delta - what's new for perl v5.6.0

DESCRIPTION

This document describes differences between the 5.005 release and the 5.6.0 release.

Core Enhancements

Interpreter cloning, threads, and concurrency

Perl 5.6.0 introduces the beginnings of support for running multiple interpreters concurrently in different threads. In conjunction with the perl_clone() API call, which can be used to selectively duplicate the state of any given interpreter, it is possible to compile a piece of code once in an interpreter, clone that interpreter one or more times, and run all the resulting interpreters in distinct threads.

On the Windows platform, this feature is used to emulate fork() at the interpreter level. See *perlfork* for details about that.

This feature is still in evolution. It is eventually meant to be used to selectively clone a subroutine and data reachable from that subroutine in a separate interpreter and run the cloned subroutine in a separate thread. Since there is no shared data between the interpreters, little or no locking will be needed (unless parts of the symbol table are explicitly shared). This is obviously intended to be an easy-to-use replacement for the existing threads support.

Support for cloning interpreters and interpreter concurrency can be enabled using the -Dusethreads Configure option (see win32/Makefile for how to enable it on Windows.) The resulting perl executable will be functionally identical to one that was built with -Dmultiplicity, but the perl_clone() API call will only be available in the former.

-Dusethreads enables the cpp macro USE_ITHREADS by default, which in turn enables Perl source code changes that provide a clear separation between the op tree and the data it operates with. The former is immutable, and can therefore be shared between an interpreter and all of its clones, while the latter is considered local to each interpreter, and is therefore copied for each clone.

Note that building Perl with the -Dusemultiplicity Configure option is adequate if you wish to run multiple **independent** interpreters concurrently in different threads. -Dusethreads only provides the additional functionality of the perl_clone() API call and other support for running **cloned** interpreters concurrently.

```
    NOTE: This is an experimental feature.  Implementation details are
    subject to change.
```

Lexically scoped warning categories

You can now control the granularity of warnings emitted by perl at a finer level using the `use warnings` pragma. *warnings* and *perllexwarn* have copious documentation on this feature.

Unicode and UTF-8 support

Perl now uses UTF-8 as its internal representation for character strings. The `utf8` and `bytes` pragmas are used to control this support in the current lexical scope. See *perlunicode*, *utf8* and *bytes* for more information.

This feature is expected to evolve quickly to support some form of I/O disciplines that can be used to specify the kind of input and output data (bytes or characters). Until that happens, additional modules from CPAN will be needed to complete the toolkit for dealing with Unicode.

```
    NOTE: This should be considered an experimental feature.
Implementation
    details are subject to change.
```

Support for interpolating named characters

The new `\N` escape interpolates named characters within strings. For example, `"Hi! \N{WHITE SMILING FACE}"` evaluates to a string with a unicode smiley face at the end.

"our" declarations

An "our" declaration introduces a value that can be best understood as a lexically scoped symbolic alias to a global variable in the package that was current where the variable was declared. This is mostly useful as an alternative to the `vars` pragma, but also provides the opportunity to introduce typing and other attributes for such variables. See *"our" in perlfunc*.

Support for strings represented as a vector of ordinals

Literals of the form `v1.2.3.4` are now parsed as a string composed of characters with the specified ordinals. This is an alternative, more readable way to construct (possibly unicode) strings instead of interpolating characters, as in `"\x{1}\x{2}\x{3}\x{4}"`. The leading `v` may be omitted if there are more than two ordinals, so `1.2.3` is parsed the same as `v1.2.3`.

Strings written in this form are also useful to represent version "numbers". It is easy to compare such version "numbers" (which are really just plain strings) using any of the usual string comparison operators `eq`, `ne`, `lt`, `gt`, etc., or perform bitwise string operations on them using `|`, `&`, etc.

In conjunction with the new $^V magic variable (which contains the perl version as a string), such literals can be used as a readable way to check if you're running a particular version of Perl:

```
# this will parse in older versions of Perl also
if ($^V and $^V gt v5.6.0) {
    # new features supported
}
```

`require` and `use` also have some special magic to support such literals, but this particular usage should be avoided because it leads to misleading error messages under versions of Perl which don't support vector strings. Using a true version number will ensure correct behavior in all versions of Perl:

```
require 5.006;    # run time check for v5.6
use 5.006_001;    # compile time check for v5.6.1
```

Also, `sprintf` and `printf` support the Perl-specific format flag `%v` to print ordinals of characters in arbitrary strings:

```
printf "v%vd", $^V;  # prints current version, such as "v5.5.650"
printf "%*vX", ":", $addr; # formats IPv6 address
printf "%*vb", " ", $bits; # displays bitstring
```

See *"Scalar value constructors" in perldata* for additional information.

Improved Perl version numbering system

Beginning with Perl version 5.6.0, the version number convention has been changed to a "dotted integer" scheme that is more commonly found in open source projects.

Maintenance versions of v5.6.0 will be released as v5.6.1, v5.6.2 etc. The next development series following v5.6.0 will be numbered v5.7.x, beginning with v5.7.0, and the next major production release following v5.6.0 will be v5.8.0.

The English module now sets $PERL_VERSION to $^V (a string value) rather than `$]` (a numeric value). (This is a potential incompatibility. Send us a report via perlbug if you are affected by this.)

The v1.2.3 syntax is also now legal in Perl. See *Support for strings represented as a vector of ordinals* for more on that.

To cope with the new versioning system's use of at least three significant digits for each version component, the method used for incrementing the subversion number has also changed slightly. We assume that versions older than v5.6.0 have been incrementing the subversion component in multiples of 10. Versions after v5.6.0 will increment them by 1. Thus, using the new notation, 5.005_03 is the "same" as v5.5.30, and the first maintenance version following v5.6.0 will be v5.6.1 (which should be read as being equivalent to a floating point value of 5.006_001 in the older format, stored in `$]`).

New syntax for declaring subroutine attributes

Formerly, if you wanted to mark a subroutine as being a method call or as requiring an automatic lock() when it is entered, you had to declare that with a `use attrs` pragma in the body of the subroutine. That can now be accomplished with declaration syntax, like this:

```
    sub mymethod : locked method ;
    ...
    sub mymethod : locked method {
...
    }

    sub othermethod :locked :method ;
    ...
    sub othermethod :locked :method {
...
    }
```

(Note how only the first `:` is mandatory, and whitespace surrounding the `:` is optional.)

AutoSplit.pm and *SelfLoader.pm* have been updated to keep the attributes with the stubs they provide. See *attributes*.

File and directory handles can be autovivified

Similar to how constructs such as `$x->[0]` autovivify a reference, handle constructors (open(), opendir(), pipe(), socketpair(), sysopen(), socket(), and accept()) now autovivify a file or directory handle if the handle passed to them is an uninitialized scalar variable. This allows the constructs such as `open(my $fh, ...)` and `open(local $fh,...)` to be used to create filehandles that will conveniently be closed automatically when the scope ends, provided there are no other references to them. This largely eliminates the need for typeglobs when opening filehandles that must be passed around, as in the following example:

```
    sub myopen {
        open my $fh, "@_"
      or die "Can't open '@_': $!";
return $fh;
    }

    {
        my $f = myopen("</etc/motd");
print <$f>;
# $f implicitly closed here
    }
```

open() with more than two arguments

If open() is passed three arguments instead of two, the second argument is used as the mode and the third argument is taken to be the file name. This is primarily useful for protecting against unintended magic behavior of the traditional two-argument form. See *"open" in perlfunc*.

64-bit support

Any platform that has 64-bit integers either

```
(1) natively as longs or ints
(2) via special compiler flags
(3) using long long or int64_t
```

is able to use "quads" (64-bit integers) as follows:

- constants (decimal, hexadecimal, octal, binary) in the code
- arguments to oct() and hex()
- arguments to print(), printf() and sprintf() (flag prefixes ll, L, q)
- printed as such
- pack() and unpack() "q" and "Q" formats
- in basic arithmetics: + - * / % (NOTE: operating close to the limits of the integer values may produce surprising results)
- in bit arithmetics: & | ^ ~ << >> (NOTE: these used to be forced to be 32 bits wide but now operate on the full native width.)
- vec()

Note that unless you have the case (a) you will have to configure and compile Perl using the -Duse64bitint Configure flag.

```
NOTE: The Configure flags -Duselonglong and -Duse64bits have been
deprecated.  Use -Duse64bitint instead.
```

There are actually two modes of 64-bitness: the first one is achieved using Configure -Duse64bitint and the second one using Configure -Duse64bitall. The difference is that the first one is minimal and the second one maximal. The first works in more places than the second.

The `use64bitint` does only as much as is required to get 64-bit integers into Perl (this may mean, for example, using "long longs") while your memory may still be limited to 2 gigabytes (because your pointers could still be 32-bit). Note that the name `64bitint` does not imply that your C compiler will be using 64-bit `ints` (it might, but it doesn't have to): the `use64bitint` means that you will be able to have 64 bits wide scalar values.

The `use64bitall` goes all the way by attempting to switch also integers (if it can), longs (and pointers) to being 64-bit. This may create an even more binary incompatible Perl than -Duse64bitint: the resulting executable may not run at all in a 32-bit box, or you may have to reboot/reconfigure/rebuild your operating system to be 64-bit aware.

Natively 64-bit systems like Alpha and Cray need neither -Duse64bitint nor -Duse64bitall.

Last but not least: note that due to Perl's habit of always using floating point numbers, the quads are still not true integers. When quads overflow their limits (0...18_446_744_073_709_551_615 unsigned, -9_223_372_036_854_775_808...9_223_372_036_854_775_807 signed), they are silently promoted to floating point numbers, after which they will start losing precision (in their lower digits).

```
NOTE: 64-bit support is still experimental on most platforms.
Existing support only covers the LP64 data model.  In particular, the
LLP64 data model is not yet supported.  64-bit libraries and system
APIs on many platforms have not stabilized--your mileage may vary.
```

Large file support

If you have filesystems that support "large files" (files larger than 2 gigabytes), you may now also be able to create and access them from Perl.

```
NOTE: The default action is to enable large file support, if
available on the platform.
```

If the large file support is on, and you have a Fcntl constant O_LARGEFILE, the O_LARGEFILE is automatically added to the flags of sysopen().

Beware that unless your filesystem also supports "sparse files" seeking to umpteen petabytes may be inadvisable.

Note that in addition to requiring a proper file system to do large files you may also need to adjust your per-process (or your per-system, or per-process-group, or per-user-group) maximum filesize limits before running Perl scripts that try to handle large files, especially if you intend to write such files.

Finally, in addition to your process/process group maximum filesize limits, you may have quota limits on your filesystems that stop you (your user id or your user group id) from using large files.

Adjusting your process/user/group/file system/operating system limits is outside the scope of Perl core language. For process limits, you may try increasing the limits using your shell's limits/limit/ulimit command before running Perl. The BSD::Resource extension (not included with the standard Perl distribution) may also be of use, it offers the getrlimit/setrlimit interface that can be used to adjust process resource usage limits, including the maximum filesize limit.

Long doubles

In some systems you may be able to use long doubles to enhance the range and precision of your double precision floating point numbers (that is, Perl's numbers). Use Configure -Duselongdouble to enable this support (if it is available).

"more bits"

You can "Configure -Dusemorebits" to turn on both the 64-bit support and the long double support.

Enhanced support for sort() subroutines

Perl subroutines with a prototype of `($$)`, and XSUBs in general, can now be used as sort subroutines. In either case, the two elements to be compared are passed as normal parameters in @_. See *"sort" in perlfunc*.

For unprototyped sort subroutines, the historical behavior of passing the elements to be compared as the global variables $a and $b remains unchanged.

sort $coderef @foo allowed

sort() did not accept a subroutine reference as the comparison function in earlier versions. This is now permitted.

File globbing implemented internally

Perl now uses the File::Glob implementation of the glob() operator automatically. This avoids using an external csh process and the problems associated with it.

```
NOTE: This is currently an experimental feature.  Interfaces and
implementation are subject to change.
```

Support for CHECK blocks

In addition to `BEGIN`, `INIT`, `END`, `DESTROY` and `AUTOLOAD`, subroutines named `CHECK` are now special. These are queued up during compilation and behave similar to END blocks, except they are called at the end of compilation rather than at the end of execution. They cannot be called directly.

POSIX character class syntax [: :] supported

For example to match alphabetic characters use /[[:alpha:]]/. See *perlre* for details.

Better pseudo-random number generator

In 5.005_0x and earlier, perl's rand() function used the C library rand(3) function. As of 5.005_52, Configure tests for drand48(), random(), and rand() (in that order) and picks the first one it finds.

These changes should result in better random numbers from rand().

Improved qw// operator

The `qw//` operator is now evaluated at compile time into a true list instead of being replaced with a run time call to `split()`. This removes the confusing misbehaviour of `qw//` in scalar context, which had inherited that behaviour from split().

Thus:

```
$foo = ($bar) = qw(a b c); print "$foo|$bar\n";
```

now correctly prints "3|a", instead of "2|a".

Better worst-case behavior of hashes

Small changes in the hashing algorithm have been implemented in order to improve the distribution of lower order bits in the hashed value. This is expected to yield better performance on keys that are repeated sequences.

pack() format 'Z' supported

The new format type 'Z' is useful for packing and unpacking null-terminated strings. See *"pack" in perlfunc*.

pack() format modifier '!' supported

The new format type modifier '!' is useful for packing and unpacking native shorts, ints, and longs. See *"pack" in perlfunc*.

pack() and unpack() support counted strings

The template character '/' can be used to specify a counted string type to be packed or unpacked. See *"pack" in perlfunc*.

Comments in pack() templates

The '#' character in a template introduces a comment up to end of the line. This facilitates documentation of pack() templates.

Weak references

In previous versions of Perl, you couldn't cache objects so as to allow them to be deleted if the last reference from outside the cache is deleted. The reference in the cache would hold a reference count on the object and the objects would never be destroyed.

Another familiar problem is with circular references. When an object references itself, its reference count would never go down to zero, and it would not get destroyed until the program is about to exit.

Weak references solve this by allowing you to "weaken" any reference, that is, make it not count towards the reference count. When the last non-weak reference to an object is deleted, the object is

destroyed and all the weak references to the object are automatically undef-ed.

To use this feature, you need the Devel::WeakRef package from CPAN, which contains additional documentation.

```
NOTE: This is an experimental feature.  Details are subject to change.
```

Binary numbers supported

Binary numbers are now supported as literals, in s?printf formats, and `oct()`:

```
$answer = 0b101010;
printf "The answer is: %b\n", oct("0b101010");
```

Lvalue subroutines

Subroutines can now return modifiable lvalues. See *"Lvalue subroutines" in perlsub*.

```
NOTE: This is an experimental feature.  Details are subject to change.
```

Some arrows may be omitted in calls through references

Perl now allows the arrow to be omitted in many constructs involving subroutine calls through references. For example, `$foo[10]->('foo')` may now be written `$foo[10]('foo')`. This is rather similar to how the arrow may be omitted from `$foo[10]->{'foo'}`. Note however, that the arrow is still required for `foo(10)->('bar')`.

Boolean assignment operators are legal lvalues

Constructs such as `($a ||= 2) += 1` are now allowed.

exists() is supported on subroutine names

The exists() builtin now works on subroutine names. A subroutine is considered to exist if it has been declared (even if implicitly). See *"exists" in perlfunc* for examples.

exists() and delete() are supported on array elements

The exists() and delete() builtins now work on simple arrays as well. The behavior is similar to that on hash elements.

exists() can be used to check whether an array element has been initialized. This avoids autovivifying array elements that don't exist. If the array is tied, the EXISTS() method in the corresponding tied package will be invoked.

delete() may be used to remove an element from the array and return it. The array element at that position returns to its uninitialized state, so that testing for the same element with exists() will return false. If the element happens to be the one at the end, the size of the array also shrinks up to the highest element that tests true for exists(), or 0 if none such is found. If the array is tied, the DELETE() method in the corresponding tied package will be invoked.

See *"exists" in perlfunc* and *"delete" in perlfunc* for examples.

Pseudo-hashes work better

Dereferencing some types of reference values in a pseudo-hash, such as `$ph->{foo}[1]`, was accidentally disallowed. This has been corrected.

When applied to a pseudo-hash element, exists() now reports whether the specified value exists, not merely if the key is valid.

delete() now works on pseudo-hashes. When given a pseudo-hash element or slice it deletes the values corresponding to the keys (but not the keys themselves). See *"Pseudo-hashes: Using an array*

as a hash" in perlref.

Pseudo-hash slices with constant keys are now optimized to array lookups at compile-time.

List assignments to pseudo-hash slices are now supported.

The `fields` pragma now provides ways to create pseudo-hashes, via fields::new() and fields::phash(). See *fields*.

```
NOTE: The pseudo-hash data type continues to be experimental.
Limiting oneself to the interface elements provided by the
fields pragma will provide protection from any future changes.
```

Automatic flushing of output buffers

fork(), exec(), system(), qx//, and pipe open()s now flush buffers of all files opened for output when the operation was attempted. This mostly eliminates confusing buffering mishaps suffered by users unaware of how Perl internally handles I/O.

This is not supported on some platforms like Solaris where a suitably correct implementation of fflush(NULL) isn't available.

Better diagnostics on meaningless filehandle operations

Constructs such as `open(<FH>)` and `close(<FH>)` are compile time errors. Attempting to read from filehandles that were opened only for writing will now produce warnings (just as writing to read-only filehandles does).

Where possible, buffered data discarded from duped input filehandle

`open(NEW, "<&OLD")` now attempts to discard any data that was previously read and buffered in `OLD` before duping the handle. On platforms where doing this is allowed, the next read operation on `NEW` will return the same data as the corresponding operation on `OLD`. Formerly, it would have returned the data from the start of the following disk block instead.

eof() has the same old magic as <>

`eof()` would return true if no attempt to read from <> had yet been made. `eof()` has been changed to have a little magic of its own, it now opens the <> files.

binmode() can be used to set :crlf and :raw modes

binmode() now accepts a second argument that specifies a discipline for the handle in question. The two pseudo-disciplines ":raw" and ":crlf" are currently supported on DOS-derivative platforms. See *"binmode" in perlfunc* and *open*.

-T filetest recognizes UTF-8 encoded files as "text"

The algorithm used for the `-T` filetest has been enhanced to correctly identify UTF-8 content as "text".

system(), backticks and pipe open now reflect exec() failure

On Unix and similar platforms, system(), qx() and open(FOO, "cmd |") etc., are implemented via fork() and exec(). When the underlying exec() fails, earlier versions did not report the error properly, since the exec() happened to be in a different process.

The child process now communicates with the parent about the error in launching the external command, which allows these constructs to return with their usual error value and set $!.

Improved diagnostics

Line numbers are no longer suppressed (under most likely circumstances) during the global destruction phase.

Diagnostics emitted from code running in threads other than the main thread are now accompanied

by the thread ID.

Embedded null characters in diagnostics now actually show up. They used to truncate the message in prior versions.

$foo::a and $foo::b are now exempt from "possible typo" warnings only if sort() is encountered in package `foo`.

Unrecognized alphabetic escapes encountered when parsing quote constructs now generate a warning, since they may take on new semantics in later versions of Perl.

Many diagnostics now report the internal operation in which the warning was provoked, like so:

```
Use of uninitialized value in concatenation (.) at (eval 1) line 1.
Use of uninitialized value in print at (eval 1) line 1.
```

Diagnostics that occur within eval may also report the file and line number where the eval is located, in addition to the eval sequence number and the line number within the evaluated text itself. For example:

```
    Not enough arguments for scalar at (eval 4)[newlib/perl5db.pl:1411]
line 2, at EOF
```

Diagnostics follow STDERR

Diagnostic output now goes to whichever file the STDERR handle is pointing at, instead of always going to the underlying C runtime library's `stderr`.

More consistent close-on-exec behavior

On systems that support a close-on-exec flag on filehandles, the flag is now set for any handles created by pipe(), socketpair(), socket(), and accept(), if that is warranted by the value of $^F that may be in effect. Earlier versions neglected to set the flag for handles created with these operators. See *"pipe" in perlfunc*, *"socketpair" in perlfunc*, *"socket" in perlfunc*, *"accept" in perlfunc*, and *"$^F" in perlvar*.

syswrite() ease-of-use

The length argument of `syswrite()` has become optional.

Better syntax checks on parenthesized unary operators

Expressions such as:

```
print defined(&foo,&bar,&baz);
print uc("foo","bar","baz");
undef($foo,&bar);
```

used to be accidentally allowed in earlier versions, and produced unpredictable behaviour. Some produced ancillary warnings when used in this way; others silently did the wrong thing.

The parenthesized forms of most unary operators that expect a single argument now ensure that they are not called with more than one argument, making the cases shown above syntax errors. The usual behaviour of:

```
print defined &foo, &bar, &baz;
print uc "foo", "bar", "baz";
undef $foo, &bar;
```

remains unchanged. See *perlop*.

Bit operators support full native integer width

The bit operators (& | ^ ~ << >>) now operate on the full native integral width (the exact size of which is available in $Config{ivsize}). For example, if your platform is either natively 64-bit or if Perl has been configured to use 64-bit integers, these operations apply to 8 bytes (as opposed to 4 bytes on 32-bit platforms). For portability, be sure to mask off the excess bits in the result of unary ~, e.g., `~$x & 0xffffffff`.

Improved security features

More potentially unsafe operations taint their results for improved security.

The `passwd` and `shell` fields returned by the getpwent(), getpwnam(), and getpwuid() are now tainted, because the user can affect their own encrypted password and login shell.

The variable modified by shmread(), and messages returned by msgrcv() (and its object-oriented interface IPC::SysV::Msg::rcv) are also tainted, because other untrusted processes can modify messages and shared memory segments for their own nefarious purposes.

More functional bareword prototype (*)

Bareword prototypes have been rationalized to enable them to be used to override builtins that accept barewords and interpret them in a special way, such as `require` or `do`.

Arguments prototyped as * will now be visible within the subroutine as either a simple scalar or as a reference to a typeglob. See *"Prototypes" in perlsub*.

require and do may be overridden

`require` and `do 'file'` operations may be overridden locally by importing subroutines of the same name into the current package (or globally by importing them into the CORE::GLOBAL:: namespace). Overriding `require` will also affect `use`, provided the override is visible at compile-time. See *"Overriding Built-in Functions" in perlsub*.

$^X variables may now have names longer than one character

Formerly, $^X was synonymous with ${"\cX"}, but $^XY was a syntax error. Now variable names that begin with a control character may be arbitrarily long. However, for compatibility reasons, these variables *must* be written with explicit braces, as `${^XY}` for example. `${^XYZ}` is synonymous with ${"\cXYZ"}. Variable names with more than one control character, such as `${^XY^Z}`, are illegal.

The old syntax has not changed. As before, `^X' may be either a literal control-X character or the two-character sequence `caret' plus `X'. When braces are omitted, the variable name stops after the control character. Thus `"$^XYZ"` continues to be synonymous with `$^X . "YZ"` as before.

As before, lexical variables may not have names beginning with control characters. As before, variables whose names begin with a control character are always forced to be in package `main'. All such variables are reserved for future extensions, except those that begin with `^_`, which may be used by user programs and are guaranteed not to acquire special meaning in any future version of Perl.

New variable $^C reflects -c switch

`$^C` has a boolean value that reflects whether perl is being run in compile-only mode (i.e. via the `-c` switch). Since BEGIN blocks are executed under such conditions, this variable enables perl code to determine whether actions that make sense only during normal running are warranted. See *perlvar*.

New variable $^V contains Perl version as a string

$^V contains the Perl version number as a string composed of characters whose ordinals match the version numbers, i.e. v5.6.0. This may be used in string comparisons.

See `Support for strings represented as a vector of ordinals` for an example.

Optional Y2K warnings

If Perl is built with the cpp macro `PERL_Y2KWARN` defined, it emits optional warnings when concatenating the number 19 with another number.

This behavior must be specifically enabled when running Configure. See *INSTALL* and *README.Y2K* .

Arrays now always interpolate into double-quoted strings

In double-quoted strings, arrays now interpolate, no matter what. The behavior in earlier versions of perl 5 was that arrays would interpolate into strings if the array had been mentioned before the string was compiled, and otherwise Perl would raise a fatal compile-time error. In versions 5.000 through 5.003, the error was

```
Literal @example now requires backslash
```

In versions 5.004_01 through 5.6.0, the error was

```
In string, @example now must be written as \@example
```

The idea here was to get people into the habit of writing `"fred\@example.com"` when they wanted a literal @ sign, just as they have always written `"Give me back my \$5"` when they wanted a literal $ sign.

Starting with 5.6.1, when Perl now sees an @ sign in a double-quoted string, it *always* attempts to interpolate an array, regardless of whether or not the array has been used or declared already. The fatal error has been downgraded to an optional warning:

```
Possible unintended interpolation of @example in string
```

This warns you that `"fred@example.com"` is going to turn into `fred.com` if you don't backslash the @. See http://www.plover.com/~mjd/perl/at-error.html for more details about the history here.

Modules and Pragmata

Modules

attributes

While used internally by Perl as a pragma, this module also provides a way to fetch subroutine and variable attributes. See *attributes*.

B

The Perl Compiler suite has been extensively reworked for this release. More of the standard Perl testsuite passes when run under the Compiler, but there is still a significant way to go to achieve production quality compiled executables.

```
    NOTE: The Compiler suite remains highly experimental.  The
    generated code may not be correct, even when it manages to
execute
    without errors.
```

Benchmark

Overall, Benchmark results exhibit lower average error and better timing accuracy.

You can now run tests for *n* seconds instead of guessing the right number of tests to run: e.g., timethese(-5, ...) will run each code for at least 5 CPU seconds. Zero as the "number of repetitions" means "for at least 3 CPU seconds". The output format has also changed. For example:

```
   use Benchmark;$x=3;timethese(-5,{a=>sub{$x*$x},b=>sub{$x**2}})
```

will now output something like this:

```
   Benchmark: running a, b, each for at least 5 CPU seconds...
             a:  5 wallclock secs ( 5.77 usr +  0.00 sys =  5.77 CPU)
@ 200551.91/s (n=1156516)
             b:  4 wallclock secs ( 5.00 usr +  0.02 sys =  5.02 CPU)
@ 159605.18/s (n=800686)
```

New features: "each for at least N CPU seconds...", "wallclock secs", and the "@ operations/CPU second (n=operations)".

timethese() now returns a reference to a hash of Benchmark objects containing the test results, keyed on the names of the tests.

timethis() now returns the iterations field in the Benchmark result object instead of 0.

timethese(), timethis(), and the new cmpthese() (see below) can also take a format specifier of 'none' to suppress output.

A new function countit() is just like timeit() except that it takes a TIME instead of a COUNT.

A new function cmpthese() prints a chart comparing the results of each test returned from a timethese() call. For each possible pair of tests, the percentage speed difference (iters/sec or seconds/iter) is shown.

For other details, see *Benchmark*.

ByteLoader

The ByteLoader is a dedicated extension to generate and run Perl bytecode. See *ByteLoader*.

constant

References can now be used.

The new version also allows a leading underscore in constant names, but disallows a double leading underscore (as in "__LINE__"). Some other names are disallowed or warned against, including BEGIN, END, etc. Some names which were forced into main:: used to fail silently in some cases; now they're fatal (outside of main::) and an optional warning (inside of main::). The ability to detect whether a constant had been set with a given name has been added.

See *constant*.

charnames

This pragma implements the `\N` string escape. See *charnames*.

Data::Dumper

A `Maxdepth` setting can be specified to avoid venturing too deeply into deep data structures. See *Data::Dumper*.

The XSUB implementation of Dump() is now automatically called if the `Useqq` setting is not in use.

Dumping `qr//` objects works correctly.

DB

`DB` is an experimental module that exposes a clean abstraction to Perl's debugging API.

DB_File

DB_File can now be built with Berkeley DB versions 1, 2 or 3. See `ext/DB_File/Changes`.

Devel::DProf

Devel::DProf, a Perl source code profiler has been added. See *Devel::DProf* and *dprofpp*.

Devel::Peek

The Devel::Peek module provides access to the internal representation of Perl variables and data. It is a data debugging tool for the XS programmer.

Dumpvalue

The Dumpvalue module provides screen dumps of Perl data.

DynaLoader

DynaLoader now supports a dl_unload_file() function on platforms that support unloading shared objects using dlclose().

Perl can also optionally arrange to unload all extension shared objects loaded by Perl. To enable this, build Perl with the Configure option `-Accflags=-DDL_UNLOAD_ALL_AT_EXIT`. (This maybe useful if you are using Apache with mod_perl.)

English

$PERL_VERSION now stands for $^V (a string value) rather than for `$]` (a numeric value).

Env

Env now supports accessing environment variables like PATH as array variables.

Fcntl

More Fcntl constants added: F_SETLK64, F_SETLKW64, O_LARGEFILE for large file (more than 4GB) access (NOTE: the O_LARGEFILE is automatically added to sysopen() flags if large file support has been configured, as is the default), Free/Net/OpenBSD locking behaviour flags F_FLOCK, F_POSIX, Linux F_SHLCK, and O_ACCMODE: the combined mask of O_RDONLY, O_WRONLY, and O_RDWR. The seek()/sysseek() constants SEEK_SET, SEEK_CUR, and SEEK_END are available via the `:seek` tag. The chmod()/stat() S_IF* constants and S_IS* functions are available via the `:mode` tag.

File::Compare

A compare_text() function has been added, which allows custom comparison functions. See *File::Compare*.

File::Find

File::Find now works correctly when the wanted() function is either autoloaded or is a symbolic reference.

A bug that caused File::Find to lose track of the working directory when pruning top-level directories has been fixed.

File::Find now also supports several other options to control its behavior. It can follow symbolic links if the `follow` option is specified. Enabling the `no_chdir` option will make File::Find skip changing the current directory when walking directories. The `untaint` flag can be useful when running with taint checks enabled.

See *File::Find*.

File::Glob

This extension implements BSD-style file globbing. By default, it will also be used for the internal implementation of the glob() operator. See *File::Glob*.

File::Spec

New methods have been added to the File::Spec module: devnull() returns the name of the null device (/dev/null on Unix) and tmpdir() the name of the temp directory (normally /tmp on Unix). There are now also methods to convert between absolute and relative filenames: abs2rel() and rel2abs(). For compatibility with operating systems that specify volume names in file paths, the splitpath(), splitdir(), and catdir() methods have been added.

File::Spec::Functions

The new File::Spec::Functions modules provides a function interface to the File::Spec module. Allows shorthand

```
    $fullname = catfile($dir1, $dir2, $file);
```

instead of

```
    $fullname = File::Spec->catfile($dir1, $dir2, $file);
```

Getopt::Long

Getopt::Long licensing has changed to allow the Perl Artistic License as well as the GPL. It used to be GPL only, which got in the way of non-GPL applications that wanted to use Getopt::Long.

Getopt::Long encourages the use of Pod::Usage to produce help messages. For example:

```
    use Getopt::Long;
    use Pod::Usage;
    my $man = 0;
    my $help = 0;
    GetOptions('help|?' => \$help, man => \$man) or pod2usage(2);
    pod2usage(1) if $help;
    pod2usage(-exitstatus => 0, -verbose => 2) if $man;

    __END__

    =head1 NAME

    sample - Using Getopt::Long and Pod::Usage

    =head1 SYNOPSIS

    sample [options] [file ...]

     Options:
       -help            brief help message
       -man             full documentation

    =head1 OPTIONS

    =over 8

    =item B<-help>

    Print a brief help message and exits.

    =item B<-man>

    Prints the manual page and exits.

    =back

    =head1 DESCRIPTION

    B<This program> will read the given input file(s) and do
something
    useful with the contents thereof.
```

```
=cut
```

See *Pod::Usage* for details.

A bug that prevented the non-option call-back <> from being specified as the first argument has been fixed.

To specify the characters < and > as option starters, use ><. Note, however, that changing option starters is strongly deprecated.

IO

write() and syswrite() will now accept a single-argument form of the call, for consistency with Perl's syswrite().

You can now create a TCP-based IO::Socket::INET without forcing a connect attempt. This allows you to configure its options (like making it non-blocking) and then call connect() manually.

A bug that prevented the IO::Socket::protocol() accessor from ever returning the correct value has been corrected.

IO::Socket::connect now uses non-blocking IO instead of alarm() to do connect timeouts.

IO::Socket::accept now uses select() instead of alarm() for doing timeouts.

IO::Socket::INET->new now sets $! correctly on failure. $@ is still set for backwards compatibility.

JPL

Java Perl Lingo is now distributed with Perl. See jpl/README for more information.

lib

`use lib` now weeds out any trailing duplicate entries. `no lib` removes all named entries.

Math::BigInt

The bitwise operations <<, >>, &, |, and ~ are now supported on bigints.

Math::Complex

The accessor methods Re, Im, arg, abs, rho, and theta can now also act as mutators (accessor $z->Re(), mutator $z->Re(3)).

The class method `display_format` and the corresponding object method `display_format`, in addition to accepting just one argument, now can also accept a parameter hash. Recognized keys of a parameter hash are `"style"`, which corresponds to the old one parameter case, and two new parameters: `"format"`, which is a printf()-style format string (defaults usually to `"%.15g"`, you can revert to the default by setting the format string to `undef`) used for both parts of a complex number, and `"polar_pretty_print"` (defaults to true), which controls whether an attempt is made to try to recognize small multiples and rationals of pi (2pi, pi/2) at the argument (angle) of a polar complex number.

The potentially disruptive change is that in list context both methods now *return the parameter hash*, instead of only the value of the `"style"` parameter.

Math::Trig

A little bit of radial trigonometry (cylindrical and spherical), radial coordinate conversions, and the great circle distance were added.

Pod::Parser, Pod::InputObjects

Pod::Parser is a base class for parsing and selecting sections of pod documentation from an input stream. This module takes care of identifying pod paragraphs and commands in the input and hands off the parsed paragraphs and commands to user-defined methods which are free to interpret or translate them as they see fit.

Pod::InputObjects defines some input objects needed by Pod::Parser, and for advanced users of Pod::Parser that need more about a command besides its name and text.

As of release 5.6.0 of Perl, Pod::Parser is now the officially sanctioned "base parser code" recommended for use by all pod2xxx translators. Pod::Text (pod2text) and Pod::Man (pod2man) have already been converted to use Pod::Parser and efforts to convert Pod::HTML (pod2html) are already underway. For any questions or comments about pod parsing and translating issues and utilities, please use the pod-people@perl.org mailing list.

For further information, please see *Pod::Parser* and *Pod::InputObjects*.

Pod::Checker, podchecker

This utility checks pod files for correct syntax, according to *perlpod*. Obvious errors are flagged as such, while warnings are printed for mistakes that can be handled gracefully. The checklist is not complete yet. See *Pod::Checker*.

Pod::ParseUtils, Pod::Find

These modules provide a set of gizmos that are useful mainly for pod translators. *Pod::Find* traverses directory structures and returns found pod files, along with their canonical names (like `File::Spec::Unix`). *Pod::ParseUtils* contains **Pod::List** (useful for storing pod list information), **Pod::Hyperlink** (for parsing the contents of `L<>` sequences) and **Pod::Cache** (for caching information about pod files, e.g., link nodes).

Pod::Select, podselect

Pod::Select is a subclass of Pod::Parser which provides a function named "podselect()" to filter out user-specified sections of raw pod documentation from an input stream. podselect is a script that provides access to Pod::Select from other scripts to be used as a filter. See *Pod::Select*.

Pod::Usage, pod2usage

Pod::Usage provides the function "pod2usage()" to print usage messages for a Perl script based on its embedded pod documentation. The pod2usage() function is generally useful to all script authors since it lets them write and maintain a single source (the pods) for documentation, thus removing the need to create and maintain redundant usage message text consisting of information already in the pods.

There is also a pod2usage script which can be used from other kinds of scripts to print usage messages from pods (even for non-Perl scripts with pods embedded in comments).

For details and examples, please see *Pod::Usage*.

Pod::Text and Pod::Man

Pod::Text has been rewritten to use Pod::Parser. While pod2text() is still available for backwards compatibility, the module now has a new preferred interface. See *Pod::Text* for the details. The new Pod::Text module is easily subclassed for tweaks to the output, and two such subclasses (Pod::Text::Termcap for man-page-style bold and underlining using termcap information, and Pod::Text::Color for markup with ANSI color sequences) are now standard.

pod2man has been turned into a module, Pod::Man, which also uses Pod::Parser. In the process, several outstanding bugs related to quotes in section headers, quoting of code escapes, and nested lists have been fixed. pod2man is now a wrapper script around this module.

SDBM_File

An EXISTS method has been added to this module (and sdbm_exists() has been added to the underlying sdbm library), so one can now call exists on an SDBM_File tied hash and get the correct result, rather than a runtime error.

A bug that may have caused data loss when more than one disk block happens to be read from the database in a single FETCH() has been fixed.

Sys::Syslog

Sys::Syslog now uses XSUBs to access facilities from syslog.h so it no longer requires syslog.ph to exist.

Sys::Hostname

Sys::Hostname now uses XSUBs to call the C library's gethostname() or uname() if they exist.

Term::ANSIColor

Term::ANSIColor is a very simple module to provide easy and readable access to the ANSI color and highlighting escape sequences, supported by most ANSI terminal emulators. It is now included standard.

Time::Local

The timelocal() and timegm() functions used to silently return bogus results when the date fell outside the machine's integer range. They now consistently croak() if the date falls in an unsupported range.

Win32

The error return value in list context has been changed for all functions that return a list of values. Previously these functions returned a list with a single element `undef` if an error occurred. Now these functions return the empty list in these situations. This applies to the following functions:

```
Win32::FsType
Win32::GetOSVersion
```

The remaining functions are unchanged and continue to return `undef` on error even in list context.

The Win32::SetLastError(ERROR) function has been added as a complement to the Win32::GetLastError() function.

The new Win32::GetFullPathName(FILENAME) returns the full absolute pathname for FILENAME in scalar context. In list context it returns a two-element list containing the fully qualified directory name and the filename. See *Win32*.

XSLoader

The XSLoader extension is a simpler alternative to DynaLoader. See *XSLoader*.

DBM Filters

A new feature called "DBM Filters" has been added to all the DBM modules--DB_File, GDBM_File, NDBM_File, ODBM_File, and SDBM_File. DBM Filters add four new methods to each DBM module:

```
filter_store_key
filter_store_value
filter_fetch_key
filter_fetch_value
```

These can be used to filter key-value pairs before the pairs are written to the database or just after they are read from the database. See *perldbmfilter* for further information.

Pragmata

`use attrs` is now obsolete, and is only provided for backward-compatibility. It's been replaced by the `sub : attributes` syntax. See *"Subroutine Attributes" in perlsub* and *attributes*.

Lexical warnings pragma, `use warnings;`, to control optional warnings. See *perllexwarn*.

`use filetest` to control the behaviour of filetests (`-r -w` ...). Currently only one subpragma

implemented, "use filetest 'access';", that uses access(2) or equivalent to check permissions instead of using stat(2) as usual. This matters in filesystems where there are ACLs (access control lists): the stat(2) might lie, but access(2) knows better.

The `open` pragma can be used to specify default disciplines for handle constructors (e.g. open()) and for qx//. The two pseudo-disciplines `:raw` and `:crlf` are currently supported on DOS-derivative platforms (i.e. where binmode is not a no-op). See also *binmode() can be used to set :crlf and :raw modes*.

Utility Changes

dprofpp

`dprofpp` is used to display profile data generated using `Devel::DProf`. See *dprofpp*.

find2perl

The `find2perl` utility now uses the enhanced features of the File::Find module. The -depth and -follow options are supported. Pod documentation is also included in the script.

h2xs

The `h2xs` tool can now work in conjunction with `C::Scan` (available from CPAN) to automatically parse real-life header files. The `-M`, `-a`, `-k`, and `-o` options are new.

perlcc

`perlcc` now supports the C and Bytecode backends. By default, it generates output from the simple C backend rather than the optimized C backend.

Support for non-Unix platforms has been improved.

perldoc

`perldoc` has been reworked to avoid possible security holes. It will not by default let itself be run as the superuser, but you may still use the **-U** switch to try to make it drop privileges first.

The Perl Debugger

Many bug fixes and enhancements were added to *perl5db.pl*, the Perl debugger. The help documentation was rearranged. New commands include `< ?`, `> ?`, and `{ ?` to list out current actions, `man` *docpage* to run your doc viewer on some perl docset, and support for quoted options. The help information was rearranged, and should be viewable once again if you're using **less** as your pager. A serious security hole was plugged--you should immediately remove all older versions of the Perl debugger as installed in previous releases, all the way back to perl3, from your system to avoid being bitten by this.

Improved Documentation

Many of the platform-specific README files are now part of the perl installation. See *perl* for the complete list.

perlapi.pod

The official list of public Perl API functions.

perlboot.pod

A tutorial for beginners on object-oriented Perl.

perlcompile.pod

An introduction to using the Perl Compiler suite.

perldbmfilter.pod

A howto document on using the DBM filter facility.

perldebug.pod

All material unrelated to running the Perl debugger, plus all low-level guts-like details that risked crushing the casual user of the debugger, have been relocated from the old manpage to the next entry below.

perldebguts.pod

This new manpage contains excessively low-level material not related to the Perl debugger, but slightly related to debugging Perl itself. It also contains some arcane internal details of how the debugging process works that may only be of interest to developers of Perl debuggers.

perlfork.pod

Notes on the fork() emulation currently available for the Windows platform.

perlfilter.pod

An introduction to writing Perl source filters.

perlhack.pod

Some guidelines for hacking the Perl source code.

perlintern.pod

A list of internal functions in the Perl source code. (List is currently empty.)

perllexwarn.pod

Introduction and reference information about lexically scoped warning categories.

perlnumber.pod

Detailed information about numbers as they are represented in Perl.

perlopentut.pod

A tutorial on using open() effectively.

perlreftut.pod

A tutorial that introduces the essentials of references.

perltootc.pod

A tutorial on managing class data for object modules.

perltodo.pod

Discussion of the most often wanted features that may someday be supported in Perl.

perlunicode.pod

An introduction to Unicode support features in Perl.

Performance enhancements

Simple sort() using { $a <=> $b } and the like are optimized

Many common sort() operations using a simple inlined block are now optimized for faster performance.

Optimized assignments to lexical variables

Certain operations in the RHS of assignment statements have been optimized to directly set the lexical variable on the LHS, eliminating redundant copying overheads.

Faster subroutine calls

Minor changes in how subroutine calls are handled internally provide marginal improvements in performance.

delete(), each(), values() and hash iteration are faster

The hash values returned by delete(), each(), values() and hashes in a list context are the actual values in the hash, instead of copies. This results in significantly better performance, because it eliminates needless copying in most situations.

Installation and Configuration Improvements

-Dusethreads means something different

The -Dusethreads flag now enables the experimental interpreter-based thread support by default. To get the flavor of experimental threads that was in 5.005 instead, you need to run Configure with "-Dusethreads -Duse5005threads".

As of v5.6.0, interpreter-threads support is still lacking a way to create new threads from Perl (i.e., `use Thread;` will not work with interpreter threads). `use Thread;` continues to be available when you specify the -Duse5005threads option to Configure, bugs and all.

```
    NOTE: Support for threads continues to be an experimental feature.
    Interfaces and implementation are subject to sudden and drastic
changes.
```

New Configure flags

The following new flags may be enabled on the Configure command line by running Configure with `-Dflag`.

```
    usemultiplicity
    usethreads useithreads (new interpreter threads: no Perl API yet)
    usethreads use5005threads (threads as they were in 5.005)

    use64bitint   (equal to now deprecated 'use64bits')
    use64bitall

    uselongdouble
    usemorebits
    uselargefiles
    usesocks   (only SOCKS v5 supported)
```

Threadedness and 64-bitness now more daring

The Configure options enabling the use of threads and the use of 64-bitness are now more daring in the sense that they no more have an explicit list of operating systems of known threads/64-bit capabilities. In other words: if your operating system has the necessary APIs and datatypes, you should be able just to go ahead and use them, for threads by Configure -Dusethreads, and for 64 bits either explicitly by Configure -Duse64bitint or implicitly if your system has 64-bit wide datatypes. See also *64-bit support*.

Long Doubles

Some platforms have "long doubles", floating point numbers of even larger range than ordinary "doubles". To enable using long doubles for Perl's scalars, use -Duselongdouble.

-Dusemorebits

You can enable both -Duse64bitint and -Duselongdouble with -Dusemorebits. See also *64-bit support*.

-Duselargefiles

Some platforms support system APIs that are capable of handling large files (typically, files larger than two gigabytes). Perl will try to use these APIs if you ask for -Duselargefiles.

See *Large file support* for more information.

installusrbinperl

You can use "Configure -Uinstallusrbinperl" which causes installperl to skip installing perl also as /usr/bin/perl. This is useful if you prefer not to modify /usr/bin for some reason or another but harmful because many scripts assume to find Perl in /usr/bin/perl.

SOCKS support

You can use "Configure -Dusesocks" which causes Perl to probe for the SOCKS proxy protocol library (v5, not v4). For more information on SOCKS, see:

```
http://www.socks.nec.com/
```

-A flag

You can "post-edit" the Configure variables using the Configure `-A` switch. The editing happens immediately after the platform specific hints files have been processed but before the actual configuration process starts. Run `Configure -h` to find out the full `-A` syntax.

Enhanced Installation Directories

The installation structure has been enriched to improve the support for maintaining multiple versions of perl, to provide locations for vendor-supplied modules, scripts, and manpages, and to ease maintenance of locally-added modules, scripts, and manpages. See the section on Installation Directories in the INSTALL file for complete details. For most users building and installing from source, the defaults should be fine.

If you previously used `Configure -Dsitelib` or `-Dsitearch` to set special values for library directories, you might wish to consider using the new `-Dsiteprefix` setting instead. Also, if you wish to re-use a config.sh file from an earlier version of perl, you should be sure to check that Configure makes sensible choices for the new directories. See INSTALL for complete details.

Platform specific changes

Supported platforms

- The Mach CThreads (NEXTSTEP, OPENSTEP) are now supported by the Thread extension.
- GNU/Hurd is now supported.
- Rhapsody/Darwin is now supported.
- EPOC is now supported (on Psion 5).
- The cygwin port (formerly cygwin32) has been greatly improved.

DOS

- Perl now works with djgpp 2.02 (and 2.03 alpha).
- Environment variable names are not converted to uppercase any more.
- Incorrect exit codes from backticks have been fixed.
- This port continues to use its own builtin globbing (not File::Glob).

OS390 (OpenEdition MVS)

Support for this EBCDIC platform has not been renewed in this release. There are difficulties in reconciling Perl's standardization on UTF-8 as its internal representation for characters with the EBCDIC character set, because the two are incompatible.

It is unclear whether future versions will renew support for this platform, but the possibility exists.

VMS

Numerous revisions and extensions to configuration, build, testing, and installation process to accommodate core changes and VMS-specific options.

Expand %ENV-handling code to allow runtime mapping to logical names, CLI symbols, and CRTL environ array.

Extension of subprocess invocation code to accept filespecs as command "verbs".

Add to Perl command line processing the ability to use default file types and to recognize Unix-style `2>&1`.

Expansion of File::Spec::VMS routines, and integration into ExtUtils::MM_VMS.

Extension of ExtUtils::MM_VMS to handle complex extensions more flexibly.

Barewords at start of Unix-syntax paths may be treated as text rather than only as logical names.

Optional secure translation of several logical names used internally by Perl.

Miscellaneous bugfixing and porting of new core code to VMS.

Thanks are gladly extended to the many people who have contributed VMS patches, testing, and ideas.

Win32

Perl can now emulate fork() internally, using multiple interpreters running in different concurrent threads. This support must be enabled at build time. See *perlfork* for detailed information.

When given a pathname that consists only of a drivename, such as `A:`, opendir() and stat() now use the current working directory for the drive rather than the drive root.

The builtin XSUB functions in the Win32:: namespace are documented. See *Win32*.

$^X now contains the full path name of the running executable.

A Win32::GetLongPathName() function is provided to complement Win32::GetFullPathName() and Win32::GetShortPathName(). See *Win32*.

POSIX::uname() is supported.

system(1,...) now returns true process IDs rather than process handles. kill() accepts any real process id, rather than strictly return values from system(1,...).

For better compatibility with Unix, `kill(0, $pid)` can now be used to test whether a process exists.

The `Shell` module is supported.

Better support for building Perl under command.com in Windows 95 has been added.

Scripts are read in binary mode by default to allow ByteLoader (and the filter mechanism in general) to work properly. For compatibility, the DATA filehandle will be set to text mode if a carriage return is detected at the end of the line containing the __END__ or __DATA__ token; if not, the DATA filehandle will be left open in binary mode. Earlier versions always opened the DATA filehandle in text mode.

The glob() operator is implemented via the `File::Glob` extension, which supports glob syntax of the C shell. This increases the flexibility of the glob() operator, but there may be compatibility issues for programs that relied on the older globbing syntax. If you want to preserve compatibility with the older syntax, you might want to run perl with `-MFile::DosGlob`. For details and compatibility information, see *File::Glob*.

Significant bug fixes

<HANDLE> on empty files

With `$/` set to `undef`, "slurping" an empty file returns a string of zero length (instead of `undef`, as it used to) the first time the HANDLE is read after `$/` is set to `undef`. Further reads yield `undef`.

This means that the following will append "foo" to an empty file (it used to do nothing):

```
perl -0777 -pi -e 's/^/foo/' empty_file
```

The behaviour of:

```
perl -pi -e 's/^/foo/' empty_file
```

is unchanged (it continues to leave the file empty).

eval '...' improvements

Line numbers (as reflected by caller() and most diagnostics) within `eval '...'` were often incorrect where here documents were involved. This has been corrected.

Lexical lookups for variables appearing in `eval '...'` within functions that were themselves called within an `eval '...'` were searching the wrong place for lexicals. The lexical search now correctly ends at the subroutine's block boundary.

The use of `return` within `eval {...}` caused $@ not to be reset correctly when no exception occurred within the eval. This has been fixed.

Parsing of here documents used to be flawed when they appeared as the replacement expression in `eval 's/.../.../e'`. This has been fixed.

All compilation errors are true errors

Some "errors" encountered at compile time were by necessity generated as warnings followed by eventual termination of the program. This enabled more such errors to be reported in a single run, rather than causing a hard stop at the first error that was encountered.

The mechanism for reporting such errors has been reimplemented to queue compile-time errors and report them at the end of the compilation as true errors rather than as warnings. This fixes cases where error messages leaked through in the form of warnings when code was compiled at run time using `eval STRING`, and also allows such errors to be reliably trapped using `eval "..."`.

Implicitly closed filehandles are safer

Sometimes implicitly closed filehandles (as when they are localized, and Perl automatically closes them on exiting the scope) could inadvertently set $? or $!. This has been corrected.

Behavior of list slices is more consistent

When taking a slice of a literal list (as opposed to a slice of an array or hash), Perl used to return an empty list if the result happened to be composed of all undef values.

The new behavior is to produce an empty list if (and only if) the original list was empty. Consider the following example:

```
@a = (1,undef,undef,2)[2,1,2];
```

The old behavior would have resulted in @a having no elements. The new behavior ensures it has three undefined elements.

Note in particular that the behavior of slices of the following cases remains unchanged:

```
@a = ()[1,2];
@a = (getpwent)[7,0];
@a = (anything_returning_empty_list())[2,1,2];
@a = @b[2,1,2];
@a = @c{'a','b','c'};
```

See *perldata*.

(\$) prototype and $foo{a}

A scalar reference prototype now correctly allows a hash or array element in that slot.

goto &sub and AUTOLOAD

The `goto &sub` construct works correctly when `&sub` happens to be autoloaded.

-bareword allowed under use integer

The autoquoting of barewords preceded by `-` did not work in prior versions when the `integer` pragma was enabled. This has been fixed.

Failures in DESTROY()

When code in a destructor threw an exception, it went unnoticed in earlier versions of Perl, unless someone happened to be looking in $@ just after the point the destructor happened to run. Such failures are now visible as warnings when warnings are enabled.

Locale bugs fixed

printf() and sprintf() previously reset the numeric locale back to the default "C" locale. This has been fixed.

Numbers formatted according to the local numeric locale (such as using a decimal comma instead of a decimal dot) caused "isn't numeric" warnings, even while the operations accessing those numbers produced correct results. These warnings have been discontinued.

Memory leaks

The `eval 'return sub {...}'` construct could sometimes leak memory. This has been fixed.

Operations that aren't filehandle constructors used to leak memory when used on invalid filehandles. This has been fixed.

Constructs that modified `@_` could fail to deallocate values in `@_` and thus leak memory. This has been corrected.

Spurious subroutine stubs after failed subroutine calls

Perl could sometimes create empty subroutine stubs when a subroutine was not found in the package. Such cases stopped later method lookups from progressing into base packages. This has been corrected.

Taint failures under -U

When running in unsafe mode, taint violations could sometimes cause silent failures. This has been fixed.

END blocks and the -c switch

Prior versions used to run BEGIN **and** END blocks when Perl was run in compile-only mode. Since this is typically not the expected behavior, END blocks are not executed anymore when the `-c` switch is used, or if compilation fails.

See *Support for CHECK blocks* for how to run things when the compile phase ends.

Potential to leak DATA filehandles

Using the __DATA__ token creates an implicit filehandle to the file that contains the token. It is the program's responsibility to close it when it is done reading from it.

This caveat is now better explained in the documentation. See *perldata*.

New or Changed Diagnostics

"%s" variable %s masks earlier declaration in same %s

(W misc) A "my" or "our" variable has been redeclared in the current scope or statement, effectively eliminating all access to the previous instance. This is almost always a typographical error. Note that the earlier variable will still exist until the end of the scope or until all closure referents to it are destroyed.

"my sub" not yet implemented

(F) Lexically scoped subroutines are not yet implemented. Don't try that yet.

"our" variable %s redeclared

(W misc) You seem to have already declared the same global once before in the current lexical scope.

'!' allowed only after types %s

(F) The '!' is allowed in pack() and unpack() only after certain types. See *"pack" in perlfunc*.

/ cannot take a count

(F) You had an unpack template indicating a counted-length string, but you have also specified an explicit size for the string. See *"pack" in perlfunc*.

/ must be followed by a, A or Z

(F) You had an unpack template indicating a counted-length string, which must be followed by one of the letters a, A or Z to indicate what sort of string is to be unpacked. See *"pack" in perlfunc*.

/ must be followed by a*, A* or Z*

(F) You had a pack template indicating a counted-length string, Currently the only things that can have their length counted are a*, A* or Z*. See *"pack" in perlfunc*.

/ must follow a numeric type

(F) You had an unpack template that contained a '#', but this did not follow some numeric unpack specification. See *"pack" in perlfunc*.

/%s/: Unrecognized escape \\%c passed through

(W regexp) You used a backslash-character combination which is not recognized by Perl. This combination appears in an interpolated variable or a '-delimited regular expression. The character was understood literally.

/%s/: Unrecognized escape \\%c in character class passed through

(W regexp) You used a backslash-character combination which is not recognized by Perl inside character classes. The character was understood literally.

/%s/ should probably be written as "%s"

(W syntax) You have used a pattern where Perl expected to find a string, as in the first argument to `join`. Perl will treat the true or false result of matching the pattern against $_ as the string, which is probably not what you had in mind.

%s() called too early to check prototype

(W prototype) You've called a function that has a prototype before the parser saw a definition

or declaration for it, and Perl could not check that the call conforms to the prototype. You need to either add an early prototype declaration for the subroutine in question, or move the subroutine definition ahead of the call to get proper prototype checking. Alternatively, if you are certain that you're calling the function correctly, you may put an ampersand before the name to avoid the warning. See *perlsub*.

%s argument is not a HASH or ARRAY element

(F) The argument to exists() must be a hash or array element, such as:

```
$foo{$bar}
$ref->{"susie"}[12]
```

%s argument is not a HASH or ARRAY element or slice

(F) The argument to delete() must be either a hash or array element, such as:

```
$foo{$bar}
$ref->{"susie"}[12]
```

or a hash or array slice, such as:

```
@foo[$bar, $baz, $xyzzy]
@{$ref->[12]}{"susie", "queue"}
```

%s argument is not a subroutine name

(F) The argument to exists() for `exists &sub` must be a subroutine name, and not a subroutine call. `exists &sub()` will generate this error.

%s package attribute may clash with future reserved word: %s

(W reserved) A lowercase attribute name was used that had a package-specific handler. That name might have a meaning to Perl itself some day, even though it doesn't yet. Perhaps you should use a mixed-case attribute name, instead. See *attributes*.

(in cleanup) %s

(W misc) This prefix usually indicates that a DESTROY() method raised the indicated exception. Since destructors are usually called by the system at arbitrary points during execution, and often a vast number of times, the warning is issued only once for any number of failures that would otherwise result in the same message being repeated.

Failure of user callbacks dispatched using the G_KEEPERR flag could also result in this warning. See *"G_KEEPERR" in perlcall*.

<> should be quotes

(F) You wrote `require <file>` when you should have written `require 'file'`.

Attempt to join self

(F) You tried to join a thread from within itself, which is an impossible task. You may be joining the wrong thread, or you may need to move the join() to some other thread.

Bad evalled substitution pattern

(F) You've used the /e switch to evaluate the replacement for a substitution, but perl found a syntax error in the code to evaluate, most likely an unexpected right brace '}'.

Bad realloc() ignored

(S) An internal routine called realloc() on something that had never been malloc()ed in the first place. Mandatory, but can be disabled by setting environment variable PERL_BADFREE to 1.

Bareword found in conditional

(W bareword) The compiler found a bareword where it expected a conditional, which often indicates that an || or && was parsed as part of the last argument of the previous construct, for example:

```
open FOO || die;
```

It may also indicate a misspelled constant that has been interpreted as a bareword:

```
use constant TYPO => 1;
if (TYOP) { print "foo" }
```

The `strict` pragma is useful in avoiding such errors.

Binary number > 0b11111111111111111111111111111111 non-portable

(W portable) The binary number you specified is larger than 2**32-1 (4294967295) and therefore non-portable between systems. See *perlport* for more on portability concerns.

Bit vector size > 32 non-portable

(W portable) Using bit vector sizes larger than 32 is non-portable.

Buffer overflow in prime_env_iter: %s

(W internal) A warning peculiar to VMS. While Perl was preparing to iterate over %ENV, it encountered a logical name or symbol definition which was too long, so it was truncated to the string shown.

Can't check filesystem of script "%s"

(P) For some reason you can't check the filesystem of the script for nosuid.

Can't declare class for non-scalar %s in "%s"

(S) Currently, only scalar variables can declared with a specific class qualifier in a "my" or "our" declaration. The semantics may be extended for other types of variables in future.

Can't declare %s in "%s"

(F) Only scalar, array, and hash variables may be declared as "my" or "our" variables. They must have ordinary identifiers as names.

Can't ignore signal CHLD, forcing to default

(W signal) Perl has detected that it is being run with the SIGCHLD signal (sometimes known as SIGCLD) disabled. Since disabling this signal will interfere with proper determination of exit status of child processes, Perl has reset the signal to its default value. This situation typically indicates that the parent program under which Perl may be running (e.g., cron) is being very careless.

Can't modify non-lvalue subroutine call

(F) Subroutines meant to be used in lvalue context should be declared as such, see *"Lvalue subroutines" in perlsub*.

Can't read CRTL environ

(S) A warning peculiar to VMS. Perl tried to read an element of %ENV from the CRTL's internal environment array and discovered the array was missing. You need to figure out where your CRTL misplaced its environ or define *PERL_ENV_TABLES* (see *perlvms*) so that environ is not searched.

Can't remove %s: %s, skipping file

(S) You requested an inplace edit without creating a backup file. Perl was unable to remove the original file to replace it with the modified file. The file was left unmodified.

Can't return %s from lvalue subroutine

(F) Perl detected an attempt to return illegal lvalues (such as temporary or readonly values) from a subroutine used as an lvalue. This is not allowed.

Can't weaken a nonreference

(F) You attempted to weaken something that was not a reference. Only references can be weakened.

Character class [:%s:] unknown

(F) The class in the character class [: :] syntax is unknown. See *perlre*.

Character class syntax [%s] belongs inside character classes

(W unsafe) The character class constructs [: :], [= =], and [. .] go *inside* character classes, the [] are part of the construct, for example: /[012[:alpha:]345]/. Note that [= =] and [. .] are not currently implemented; they are simply placeholders for future extensions.

Constant is not %s reference

(F) A constant value (perhaps declared using the `use constant` pragma) is being dereferenced, but it amounts to the wrong type of reference. The message indicates the type of reference that was expected. This usually indicates a syntax error in dereferencing the constant value. See *"Constant Functions" in perlsub* and *constant*.

constant(%s): %s

(F) The parser found inconsistencies either while attempting to define an overloaded constant, or when trying to find the character name specified in the `\N{...}` escape. Perhaps you forgot to load the corresponding `overload` or `charnames` pragma? See *charnames* and *overload*.

CORE::%s is not a keyword

(F) The CORE:: namespace is reserved for Perl keywords.

defined(@array) is deprecated

(D) defined() is not usually useful on arrays because it checks for an undefined *scalar* value. If you want to see if the array is empty, just use `if (@array) { # not empty }` for example.

defined(%hash) is deprecated

(D) defined() is not usually useful on hashes because it checks for an undefined *scalar* value. If you want to see if the hash is empty, just use `if (%hash) { # not empty }` for example.

Did not produce a valid header

See Server error.

(Did you mean "local" instead of "our"?)

(W misc) Remember that "our" does not localize the declared global variable. You have declared it again in the same lexical scope, which seems superfluous.

Document contains no data

See Server error.

entering effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

false [] range "%s" in regexp

(W regexp) A character class range must start and end at a literal character, not another

character class like `\d` or `[:alpha:]`. The "-" in your false range is interpreted as a literal "-". Consider quoting the "-", "\-". See *perlre*.

Filehandle %s opened only for output

(W io) You tried to read from a filehandle opened only for writing. If you intended it to be a read/write filehandle, you needed to open it with "+<" or "+>" or "+>>" instead of with "<" or nothing. If you intended only to read from the file, use "<". See *"open" in perlfunc*.

flock() on closed filehandle %s

(W closed) The filehandle you're attempting to flock() got itself closed some time before now. Check your logic flow. flock() operates on filehandles. Are you attempting to call flock() on a dirhandle by the same name?

Global symbol "%s" requires explicit package name

(F) You've said "use strict vars", which indicates that all variables must either be lexically scoped (using "my"), declared beforehand using "our", or explicitly qualified to say which package the global variable is in (using "::").

Hexadecimal number > 0xffffffff non-portable

(W portable) The hexadecimal number you specified is larger than 2**32-1 (4294967295) and therefore non-portable between systems. See *perlport* for more on portability concerns.

Ill-formed CRTL environ value "%s"

(W internal) A warning peculiar to VMS. Perl tried to read the CRTL's internal environ array, and encountered an element without the = delimiter used to separate keys from values. The element is ignored.

Ill-formed message in prime_env_iter: |%s|

(W internal) A warning peculiar to VMS. Perl tried to read a logical name or CLI symbol definition when preparing to iterate over %ENV, and didn't see the expected delimiter between key and value, so the line was ignored.

Illegal binary digit %s

(F) You used a digit other than 0 or 1 in a binary number.

Illegal binary digit %s ignored

(W digit) You may have tried to use a digit other than 0 or 1 in a binary number. Interpretation of the binary number stopped before the offending digit.

Illegal number of bits in vec

(F) The number of bits in vec() (the third argument) must be a power of two from 1 to 32 (or 64, if your platform supports that).

Integer overflow in %s number

(W overflow) The hexadecimal, octal or binary number you have specified either as a literal or as an argument to hex() or oct() is too big for your architecture, and has been converted to a floating point number. On a 32-bit architecture the largest hexadecimal, octal or binary number representable without overflow is 0xFFFFFFFF, 037777777777, or 0b11111111111111111111111111111111 respectively. Note that Perl transparently promotes all numbers to a floating point representation internally--subject to loss of precision errors in subsequent operations.

Invalid %s attribute: %s

The indicated attribute for a subroutine or variable was not recognized by Perl or by a user-supplied handler. See *attributes*.

Invalid %s attributes: %s

The indicated attributes for a subroutine or variable were not recognized by Perl or by a user-supplied handler. See *attributes*.

invalid [] range "%s" in regexp

The offending range is now explicitly displayed.

Invalid separator character %s in attribute list

(F) Something other than a colon or whitespace was seen between the elements of an attribute list. If the previous attribute had a parenthesised parameter list, perhaps that list was terminated too soon. See *attributes*.

Invalid separator character %s in subroutine attribute list

(F) Something other than a colon or whitespace was seen between the elements of a subroutine attribute list. If the previous attribute had a parenthesised parameter list, perhaps that list was terminated too soon.

leaving effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

Lvalue subs returning %s not implemented yet

(F) Due to limitations in the current implementation, array and hash values cannot be returned in subroutines used in lvalue context. See *"Lvalue subroutines" in perlsub*.

Method %s not permitted

See Server error.

Missing %sbrace%s on \N{}

(F) Wrong syntax of character name literal `\N{charname}` within double-quotish context.

Missing command in piped open

(W pipe) You used the `open(FH, "| command")` or `open(FH, "command |")` construction, but the command was missing or blank.

Missing name in "my sub"

(F) The reserved syntax for lexically scoped subroutines requires that they have a name with which they can be found.

No %s specified for -%c

(F) The indicated command line switch needs a mandatory argument, but you haven't specified one.

No package name allowed for variable %s in "our"

(F) Fully qualified variable names are not allowed in "our" declarations, because that doesn't make much sense under existing semantics. Such syntax is reserved for future extensions.

No space allowed after -%c

(F) The argument to the indicated command line switch must follow immediately after the switch, without intervening spaces.

no UTC offset information; assuming local time is UTC

(S) A warning peculiar to VMS. Perl was unable to find the local timezone offset, so it's assuming that local system time is equivalent to UTC. If it's not, define the logical name *SYS$TIMEZONE_DIFFERENTIAL* to translate to the number of seconds which need to be added to UTC to get local time.

Octal number > 037777777777 non-portable

(W portable) The octal number you specified is larger than 2**32-1 (4294967295) and therefore non-portable between systems. See *perlport* for more on portability concerns.

See also *perlport* for writing portable code.

panic: del_backref

(P) Failed an internal consistency check while trying to reset a weak reference.

panic: kid popen errno read

(F) forked child returned an incomprehensible message about its errno.

panic: magic_killbackrefs

(P) Failed an internal consistency check while trying to reset all weak references to an object.

Parentheses missing around "%s" list

(W parenthesis) You said something like

```
my $foo, $bar = @_;
```

when you meant

```
my ($foo, $bar) = @_;
```

Remember that "my", "our", and "local" bind tighter than comma.

Possible unintended interpolation of %s in string

(W ambiguous) It used to be that Perl would try to guess whether you wanted an array interpolated or a literal @. It no longer does this; arrays are now *always* interpolated into strings. This means that if you try something like:

```
print "fred@example.com";
```

and the array `@example` doesn't exist, Perl is going to print `fred.com`, which is probably not what you wanted. To get a literal @ sign in a string, put a backslash before it, just as you would to get a literal $ sign.

Possible Y2K bug: %s

(W y2k) You are concatenating the number 19 with another number, which could be a potential Year 2000 problem.

pragma "attrs" is deprecated, use "sub NAME : ATTRS" instead

(W deprecated) You have written something like this:

```
sub doit
{
    use attrs qw(locked);
}
```

You should use the new declaration syntax instead.

```
sub doit : locked
{
    ...
```

The `use attrs` pragma is now obsolete, and is only provided for backward-compatibility. See *"Subroutine Attributes" in perlsub*.

Premature end of script headers

See Server error.

Repeat count in pack overflows

(F) You can't specify a repeat count so large that it overflows your signed integers. See *"pack" in perlfunc*.

Repeat count in unpack overflows

(F) You can't specify a repeat count so large that it overflows your signed integers. See *"unpack" in perlfunc*.

realloc() of freed memory ignored

(S) An internal routine called realloc() on something that had already been freed.

Reference is already weak

(W misc) You have attempted to weaken a reference that is already weak. Doing so has no effect.

setpgrp can't take arguments

(F) Your system has the setpgrp() from BSD 4.2, which takes no arguments, unlike POSIX setpgid(), which takes a process ID and process group ID.

Strange *+?{} on zero-length expression

(W regexp) You applied a regular expression quantifier in a place where it makes no sense, such as on a zero-width assertion. Try putting the quantifier inside the assertion instead. For example, the way to match "abc" provided that it is followed by three repetitions of "xyz" is `/abc(?=(?:xyz){3})/`, not `/abc(?=xyz){3}/`.

switching effective %s is not implemented

(F) While under the `use filetest` pragma, we cannot switch the real and effective uids or gids.

This Perl can't reset CRTL environ elements (%s)

This Perl can't set CRTL environ elements (%s=%s)

(W internal) Warnings peculiar to VMS. You tried to change or delete an element of the CRTL's internal environ array, but your copy of Perl wasn't built with a CRTL that contained the setenv() function. You'll need to rebuild Perl with a CRTL that does, or redefine *PERL_ENV_TABLES* (see *perlvms*) so that the environ array isn't the target of the change to %ENV which produced the warning.

Too late to run %s block

(W void) A CHECK or INIT block is being defined during run time proper, when the opportunity to run them has already passed. Perhaps you are loading a file with `require` or `do` when you should be using `use` instead. Or perhaps you should put the `require` or `do` inside a BEGIN block.

Unknown open() mode '%s'

(F) The second argument of 3-argument open() is not among the list of valid modes: <, >, >>, +<, +>, +>>, `-|`, `|-`.

Unknown process %x sent message to prime_env_iter: %s

(P) An error peculiar to VMS. Perl was reading values for %ENV before iterating over it, and someone else stuck a message in the stream of data Perl expected. Someone's very confused, or perhaps trying to subvert Perl's population of %ENV for nefarious purposes.

Unrecognized escape \\%c passed through

(W misc) You used a backslash-character combination which is not recognized by Perl. The character was understood literally.

Unterminated attribute parameter in attribute list

(F) The lexer saw an opening (left) parenthesis character while parsing an attribute list, but the matching closing (right) parenthesis character was not found. You may need to add (or remove) a backslash character to get your parentheses to balance. See *attributes*.

Unterminated attribute list

(F) The lexer found something other than a simple identifier at the start of an attribute, and it wasn't a semicolon or the start of a block. Perhaps you terminated the parameter list of the previous attribute too soon. See *attributes*.

Unterminated attribute parameter in subroutine attribute list

(F) The lexer saw an opening (left) parenthesis character while parsing a subroutine attribute list, but the matching closing (right) parenthesis character was not found. You may need to add (or remove) a backslash character to get your parentheses to balance.

Unterminated subroutine attribute list

(F) The lexer found something other than a simple identifier at the start of a subroutine attribute, and it wasn't a semicolon or the start of a block. Perhaps you terminated the parameter list of the previous attribute too soon.

Value of CLI symbol "%s" too long

(W misc) A warning peculiar to VMS. Perl tried to read the value of an %ENV element from a CLI symbol table, and found a resultant string longer than 1024 characters. The return value has been truncated to 1024 characters.

Version number must be a constant number

(P) The attempt to translate a `use Module n.n LIST` statement into its equivalent `BEGIN` block found an internal inconsistency with the version number.

New tests

lib/attrs

Compatibility tests for `sub : attrs` vs the older `use attrs`.

lib/env

Tests for new environment scalar capability (e.g., `use Env qw($BAR);`).

lib/env-array

Tests for new environment array capability (e.g., `use Env qw(@PATH);`).

lib/io_const

IO constants (SEEK_*, _IO*).

lib/io_dir

Directory-related IO methods (new, read, close, rewind, tied delete).

lib/io_multihomed

INET sockets with multi-homed hosts.

lib/io_poll

IO poll().

lib/io_unix

UNIX sockets.

op/attrs

Regression tests for `my ($x,@y,%z) : attrs` and <sub : attrs>.

op/filetest

File test operators.

op/lex_assign

Verify operations that access pad objects (lexicals and temporaries).

op/exists_sub

Verify `exists &sub` operations.

Incompatible Changes

Perl Source Incompatibilities

Beware that any new warnings that have been added or old ones that have been enhanced are **not** considered incompatible changes.

Since all new warnings must be explicitly requested via the `-w` switch or the `warnings` pragma, it is ultimately the programmer's responsibility to ensure that warnings are enabled judiciously.

CHECK is a new keyword

All subroutine definitions named CHECK are now special. See `/"Support for CHECK blocks"` for more information.

Treatment of list slices of undef has changed

There is a potential incompatibility in the behavior of list slices that are comprised entirely of undefined values. See *Behavior of list slices is more consistent*.

Format of $English::PERL_VERSION is different

The English module now sets $PERL_VERSION to $^V (a string value) rather than `$]` (a numeric value). This is a potential incompatibility. Send us a report via perlbug if you are affected by this.

See *Improved Perl version numbering system* for the reasons for this change.

Literals of the form `1.2.3` parse differently

Previously, numeric literals with more than one dot in them were interpreted as a floating point number concatenated with one or more numbers. Such "numbers" are now parsed as strings composed of the specified ordinals.

For example, `print 97.98.99` used to output `97.9899` in earlier versions, but now prints `abc`.

See *Support for strings represented as a vector of ordinals*.

Possibly changed pseudo-random number generator

Perl programs that depend on reproducing a specific set of pseudo-random numbers may now produce different output due to improvements made to the rand() builtin. You can use `sh Configure -Drandfunc=rand` to obtain the old behavior.

See *Better pseudo-random number generator*.

Hashing function for hash keys has changed

Even though Perl hashes are not order preserving, the apparently random order encountered when iterating on the contents of a hash is actually determined by the hashing algorithm used. Improvements in the algorithm may yield a random order that is **different** from that of previous versions, especially when iterating on hashes.

See *Better worst-case behavior of hashes* for additional information.

`undef` fails on read only values

Using the `undef` operator on a readonly value (such as $1) has the same effect as assigning `undef` to the readonly value--it throws an exception.

Close-on-exec bit may be set on pipe and socket handles

Pipe and socket handles are also now subject to the close-on-exec behavior determined by the special variable $^F.

See *More consistent close-on-exec behavior*.

Writing `"$$1"` to mean `"${$}1"` is unsupported

Perl 5.004 deprecated the interpretation of `$$1` and similar within interpolated strings to mean `$$ . "1"`, but still allowed it.

In Perl 5.6.0 and later, `"$$1"` always means `"${$1}"`.

delete(), each(), values() and `\(%h)`

operate on aliases to values, not copies

delete(), each(), values() and hashes (e.g. `\(%h)`) in a list context return the actual values in the hash, instead of copies (as they used to in earlier versions). Typical idioms for using these constructs copy the returned values, but this can make a significant difference when creating references to the returned values. Keys in the hash are still returned as copies when iterating on a hash.

See also *delete(), each(), values() and hash iteration are faster*.

vec(EXPR,OFFSET,BITS) enforces powers-of-two BITS

vec() generates a run-time error if the BITS argument is not a valid power-of-two integer.

Text of some diagnostic output has changed

Most references to internal Perl operations in diagnostics have been changed to be more descriptive. This may be an issue for programs that may incorrectly rely on the exact text of diagnostics for proper functioning.

%@ has been removed

The undocumented special variable %@ that used to accumulate "background" errors (such as those that happen in DESTROY()) has been removed, because it could potentially result in memory leaks.

Parenthesized not() behaves like a list operator

The `not` operator now falls under the "if it looks like a function, it behaves like a function" rule.

As a result, the parenthesized form can be used with `grep` and `map`. The following construct used to be a syntax error before, but it works as expected now:

```
grep not($_), @things;
```

On the other hand, using `not` with a literal list slice may not work. The following previously allowed construct:

```
print not (1,2,3)[0];
```

needs to be written with additional parentheses now:

```
print not((1,2,3)[0]);
```

The behavior remains unaffected when `not` is not followed by parentheses.

Semantics of bareword prototype `(*)` have changed

The semantics of the bareword prototype `*` have changed. Perl 5.005 always coerced simple

scalar arguments to a typeglob, which wasn't useful in situations where the subroutine must distinguish between a simple scalar and a typeglob. The new behavior is to not coerce bareword arguments to a typeglob. The value will always be visible as either a simple scalar or as a reference to a typeglob.

See *More functional bareword prototype (*)*.

Semantics of bit operators may have changed on 64-bit platforms

If your platform is either natively 64-bit or if Perl has been configured to used 64-bit integers, i.e., $Config{ivsize} is 8, there may be a potential incompatibility in the behavior of bitwise numeric operators (& | ^ ~ << >>). These operators used to strictly operate on the lower 32 bits of integers in previous versions, but now operate over the entire native integral width. In particular, note that unary ~ will produce different results on platforms that have different $Config{ivsize}. For portability, be sure to mask off the excess bits in the result of unary ~, e.g., `~$x & 0xffffffff`.

See *Bit operators support full native integer width*.

More builtins taint their results

As described in *Improved security features*, there may be more sources of taint in a Perl program.

To avoid these new tainting behaviors, you can build Perl with the Configure option `-Accflags=-DINCOMPLETE_TAINTS`. Beware that the ensuing perl binary may be insecure.

C Source Incompatibilities

`PERL_POLLUTE`

Release 5.005 grandfathered old global symbol names by providing preprocessor macros for extension source compatibility. As of release 5.6.0, these preprocessor definitions are not available by default. You need to explicitly compile perl with `-DPERL_POLLUTE` to get these definitions. For extensions still using the old symbols, this option can be specified via MakeMaker:

```
perl Makefile.PL POLLUTE=1
```

`PERL_IMPLICIT_CONTEXT`

This new build option provides a set of macros for all API functions such that an implicit interpreter/thread context argument is passed to every API function. As a result of this, something like `sv_setsv(foo,bar)` amounts to a macro invocation that actually translates to something like `Perl_sv_setsv(my_perl,foo,bar)`. While this is generally expected to not have any significant source compatibility issues, the difference between a macro and a real function call will need to be considered.

This means that there **is** a source compatibility issue as a result of this if your extensions attempt to use pointers to any of the Perl API functions.

Note that the above issue is not relevant to the default build of Perl, whose interfaces continue to match those of prior versions (but subject to the other options described here).

See *"The Perl API" in perlguts* for detailed information on the ramifications of building Perl with this option.

```
    NOTE: PERL_IMPLICIT_CONTEXT is automatically enabled whenever
Perl is built
    with one of -Dusethreads, -Dusemultiplicity, or both.  It is not
    intended to be enabled by users at this time.
```

`PERL_POLLUTE_MALLOC`

Enabling Perl's malloc in release 5.005 and earlier caused the namespace of the system's malloc family of functions to be usurped by the Perl versions, since by default they used the

same names. Besides causing problems on platforms that do not allow these functions to be cleanly replaced, this also meant that the system versions could not be called in programs that used Perl's malloc. Previous versions of Perl have allowed this behaviour to be suppressed with the HIDEMYMALLOC and EMBEDMYMALLOC preprocessor definitions.

As of release 5.6.0, Perl's malloc family of functions have default names distinct from the system versions. You need to explicitly compile perl with `-DPERL_POLLUTE_MALLOC` to get the older behaviour. HIDEMYMALLOC and EMBEDMYMALLOC have no effect, since the behaviour they enabled is now the default.

Note that these functions do **not** constitute Perl's memory allocation API. See *"Memory Allocation" in perlguts* for further information about that.

Compatible C Source API Changes

`PATCHLEVEL` is now `PERL_VERSION`

The cpp macros `PERL_REVISION`, `PERL_VERSION`, and `PERL_SUBVERSION` are now available by default from perl.h, and reflect the base revision, patchlevel, and subversion respectively. `PERL_REVISION` had no prior equivalent, while `PERL_VERSION` and `PERL_SUBVERSION` were previously available as `PATCHLEVEL` and `SUBVERSION`.

The new names cause less pollution of the **cpp** namespace and reflect what the numbers have come to stand for in common practice. For compatibility, the old names are still supported when *patchlevel.h* is explicitly included (as required before), so there is no source incompatibility from the change.

Binary Incompatibilities

In general, the default build of this release is expected to be binary compatible for extensions built with the 5.005 release or its maintenance versions. However, specific platforms may have broken binary compatibility due to changes in the defaults used in hints files. Therefore, please be sure to always check the platform-specific README files for any notes to the contrary.

The usethreads or usemultiplicity builds are **not** binary compatible with the corresponding builds in 5.005.

On platforms that require an explicit list of exports (AIX, OS/2 and Windows, among others), purely internal symbols such as parser functions and the run time opcodes are not exported by default. Perl 5.005 used to export all functions irrespective of whether they were considered part of the public API or not.

For the full list of public API functions, see *perlapi*.

Known Problems

Thread test failures

The subtests 19 and 20 of lib/thr5005.t test are known to fail due to fundamental problems in the 5.005 threading implementation. These are not new failures--Perl 5.005_0x has the same bugs, but didn't have these tests.

EBCDIC platforms not supported

In earlier releases of Perl, EBCDIC environments like OS390 (also known as Open Edition MVS) and VM-ESA were supported. Due to changes required by the UTF-8 (Unicode) support, the EBCDIC platforms are not supported in Perl 5.6.0.

In 64-bit HP-UX the lib/io_multihomed test may hang

The lib/io_multihomed test may hang in HP-UX if Perl has been configured to be 64-bit. Because other 64-bit platforms do not hang in this test, HP-UX is suspect. All other tests pass in 64-bit HP-UX. The test attempts to create and connect to "multihomed" sockets (sockets which have multiple IP addresses).

NEXTSTEP 3.3 POSIX test failure

In NEXTSTEP 3.3p2 the implementation of the strftime(3) in the operating system libraries is buggy: the %j format numbers the days of a month starting from zero, which, while being logical to programmers, will cause the subtests 19 to 27 of the lib/posix test may fail.

Tru64 (aka Digital UNIX, aka DEC OSF/1) lib/sdbm test failure with gcc

If compiled with gcc 2.95 the lib/sdbm test will fail (dump core). The cure is to use the vendor cc, it comes with the operating system and produces good code.

UNICOS/mk CC failures during Configure run

In UNICOS/mk the following errors may appear during the Configure run:

```
 Guessing which symbols your C compiler and preprocessor define...
 CC-20 cc: ERROR File = try.c, Line = 3
 ...
   bad switch yylook 79bad switch yylook 79bad switch yylook 79bad switch
yylook 79#ifdef A29K
 ...
 4 errors detected in the compilation of "try.c".
```

The culprit is the broken awk of UNICOS/mk. The effect is fortunately rather mild: Perl itself is not adversely affected by the error, only the h2ph utility coming with Perl, and that is rather rarely needed these days.

Arrow operator and arrays

When the left argument to the arrow operator `->` is an array, or the `scalar` operator operating on an array, the result of the operation must be considered erroneous. For example:

```
    @x->[2]
    scalar(@x)->[2]
```

These expressions will get run-time errors in some future release of Perl.

Experimental features

As discussed above, many features are still experimental. Interfaces and implementation of these features are subject to change, and in extreme cases, even subject to removal in some future release of Perl. These features include the following:

Threads

Unicode

64-bit support

Lvalue subroutines

Weak references

The pseudo-hash data type

The Compiler suite

Internal implementation of file globbing

The DB module

The regular expression code constructs:

`(?{ code })` and `(??{ code })`

Obsolete Diagnostics

Character class syntax [: :] is reserved for future extensions

(W) Within regular expression character classes ([]) the syntax beginning with "[:" and ending with ":]" is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: "\[:" and ":\]".

Ill-formed logical name |%s| in prime_env_iter

(W) A warning peculiar to VMS. A logical name was encountered when preparing to iterate over %ENV which violates the syntactic rules governing logical names. Because it cannot be translated normally, it is skipped, and will not appear in %ENV. This may be a benign occurrence, as some software packages might directly modify logical name tables and introduce nonstandard names, or it may indicate that a logical name table has been corrupted.

In string, @%s now must be written as \@%s

The description of this error used to say:

```
(Someday it will simply assume that an unbackslashed @
 interpolates an array.)
```

That day has come, and this fatal error has been removed. It has been replaced by a non-fatal warning instead. See *Arrays now always interpolate into double-quoted strings* for details.

Probable precedence problem on %s

(W) The compiler found a bareword where it expected a conditional, which often indicates that an || or && was parsed as part of the last argument of the previous construct, for example:

```
open FOO || die;
```

regexp too big

(F) The current implementation of regular expressions uses shorts as address offsets within a string. Unfortunately this means that if the regular expression compiles to longer than 32767, it'll blow up. Usually when you want a regular expression this big, there is a better way to do it with multiple statements. See *perlre*.

Use of "$$<digit>" to mean "${$}<digit>" is deprecated

(D) Perl versions before 5.004 misinterpreted any type marker followed by "$" and a digit. For example, "$$0" was incorrectly taken to mean "${$}0" instead of "${$0}". This bug is (mostly) fixed in Perl 5.004.

However, the developers of Perl 5.004 could not fix this bug completely, because at least two widely-used modules depend on the old meaning of "$$0" in a string. So Perl 5.004 still interprets "$$<digit>" in the old (broken) way inside strings; but it generates this message as a warning. And in Perl 5.005, this special treatment will cease.

Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup. There may also be information at http://www.perl.com/perl/ , the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to perlbug@perl.org to be analysed by the Perl porting team.

SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

HISTORY

Written by Gurusamy Sarathy <*gsar@activestate.com*>, with many contributions from The Perl Porters.

Send omissions or corrections to <*perlbug@perl.org*>.