NAME

perl5004delta - what's new for perl5.004

DESCRIPTION

This document describes differences between the 5.003 release (as documented in *Programming Perl*, second edition--the Camel Book) and this one.

Supported Environments

Perl5.004 builds out of the box on Unix, Plan 9, LynxOS, VMS, OS/2, QNX, AmigaOS, and Windows NT. Perl runs on Windows 95 as well, but it cannot be built there, for lack of a reasonable command interpreter.

Core Changes

Most importantly, many bugs were fixed, including several security problems. See the *Changes* file in the distribution for details.

List assignment to %ENV works

`%ENV = ()` and `%ENV = @list` now work as expected (except on VMS where it generates a fatal error).

Change to "Can't locate Foo.pm in @INC" error

The error "Can't locate Foo.pm in @INC" now lists the contents of @INC for easier debugging.

Compilation option: Binary compatibility with 5.003

There is a new Configure question that asks if you want to maintain binary compatibility with Perl 5.003. If you choose binary compatibility, you do not have to recompile your extensions, but you might have symbol conflicts if you embed Perl in another application, just as in the 5.003 release. By default, binary compatibility is preserved at the expense of symbol table pollution.

$PERL5OPT environment variable

You may now put Perl options in the $PERL5OPT environment variable. Unless Perl is running with taint checks, it will interpret this variable as if its contents had appeared on a "#!perl" line at the beginning of your script, except that hyphens are optional. PERL5OPT may only be used to set the following switches: **-[DIMUdmw]**.

Limitations on -M, -m, and -T options

The `-M` and `-m` options are no longer allowed on the `#!` line of a script. If a script needs a module, it should invoke it with the `use` pragma.

The **-T** option is also forbidden on the `#!` line of a script, unless it was present on the Perl command line. Due to the way `#!` works, this usually means that **-T** must be in the first argument. Thus:

```
#!/usr/bin/perl -T -w
```

will probably work for an executable script invoked as `scriptname`, while:

```
#!/usr/bin/perl -w -T
```

will probably fail under the same conditions. (Non-Unix systems will probably not follow this rule.) But `perl scriptname` is guaranteed to fail, since then there is no chance of **-T** being found on the command line before it is found on the `#!` line.

More precise warnings

If you removed the **-w** option from your Perl 5.003 scripts because it made Perl too verbose, we recommend that you try putting it back when you upgrade to Perl 5.004. Each new perl version tends to remove some undesirable warnings, while adding new warnings that may catch bugs in your

Deprecated inherited AUTOLOAD for non-methods

Before Perl 5.004, AUTOLOAD functions were looked up as methods (using the @ISA hierarchy), even when the function to be autoloaded was called as a plain function (e.g. `Foo::bar()`), not a method (e.g. `Foo->bar()` or `$obj->bar()`).

Perl 5.005 will use method lookup only for methods' AUTOLOADs. However, there is a significant base of existing code that may be using the old behavior. So, as an interim step, Perl 5.004 issues an optional warning when a non-method uses an inherited AUTOLOAD.

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

Previously deprecated %OVERLOAD is no longer usable

Using %OVERLOAD to define overloading was deprecated in 5.003. Overloading is now defined using the overload pragma. %OVERLOAD is still used internally but should not be used by Perl scripts. See *overload* for more details.

Subroutine arguments created only when they're modified

In Perl 5.004, nonexistent array and hash elements used as subroutine parameters are brought into existence only if they are actually assigned to (via @_).

Earlier versions of Perl vary in their handling of such arguments. Perl versions 5.002 and 5.003 always brought them into existence. Perl versions 5.000 and 5.001 brought them into existence only if they were not the first argument (which was almost certainly a bug). Earlier versions of Perl never brought them into existence.

For example, given this code:

```
undef @a; undef %a;
sub show { print $_[0] };
sub change { $_[0]++ };
show($a[2]);
change($a{b});
```

After this code executes in Perl 5.004, $a{b} exists but $a[2] does not. In Perl 5.002 and 5.003, both $a{b} and $a[2] would have existed (but $a[2]'s value would have been undefined).

Group vector changeable with $)

The `$)` special variable has always (well, in Perl 5, at least) reflected not only the current effective group, but also the group list as returned by the `getgroups()` C function (if there is one). However, until this release, there has not been a way to call the `setgroups()` C function from Perl.

In Perl 5.004, assigning to `$)` is exactly symmetrical with examining it: The first number in its string value is used as the effective gid; if there are any numbers after the first one, they are passed to the `setgroups()` C function (if there is one).

Fixed parsing of $$<digit>, &$<digit>, etc.

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.

Fixed localization of $<digit>, $&, etc.

Perl versions before 5.004 did not always properly localize the regex-related special variables. Perl 5.004 does localize them, as the documentation has always said it should. This may result in $1, $2, etc. no longer being set where existing programs use them.

No resetting of $. on implicit close

The documentation for Perl 5.0 has always stated that `$.` is *not* reset when an already-open file handle is reopened with no intervening call to `close`. Due to a bug, perl versions 5.000 through 5.003 *did* reset `$.` under that circumstance; Perl 5.004 does not.

wantarray may return undef

The `wantarray` operator returns true if a subroutine is expected to return a list, and false otherwise. In Perl 5.004, `wantarray` can also return the undefined value if a subroutine's return value will not be used at all, which allows subroutines to avoid a time-consuming calculation of a return value if it isn't going to be used.

eval EXPR determines value of EXPR in scalar context

Perl (version 5) used to determine the value of EXPR inconsistently, sometimes incorrectly using the surrounding context for the determination. Now, the value of EXPR (before being parsed by eval) is always determined in a scalar context. Once parsed, it is executed as before, by providing the context that the scope surrounding the eval provided. This change makes the behavior Perl4 compatible, besides fixing bugs resulting from the inconsistent behavior. This program:

```
@a = qw(time now is time);
print eval @a;
print '|', scalar eval @a;
```

used to print something like "timenowis881399109|4", but now (and in perl4) prints "4|4".

Changes to tainting checks

A bug in previous versions may have failed to detect some insecure conditions when taint checks are turned on. (Taint checks are used in setuid or setgid scripts, or when explicitly turned on with the `-T` invocation option.) Although it's unlikely, this may cause a previously-working script to now fail, which should be construed as a blessing since that indicates a potentially-serious security hole was just plugged.

The new restrictions when tainting include:

No glob() or <*>

> These operators may spawn the C shell (csh), which cannot be made safe. This restriction will be lifted in a future version of Perl when globbing is implemented without the use of an external program.

No spawning if tainted $CDPATH, $ENV, $BASH_ENV

> These environment variables may alter the behavior of spawned programs (especially shells) in ways that subvert security. So now they are treated as dangerous, in the manner of $IFS and $PATH.

No spawning if tainted $TERM doesn't look like a terminal name

> Some termcap libraries do unsafe things with $TERM. However, it would be unnecessarily harsh to treat all $TERM values as unsafe, since only shell metacharacters can cause trouble in $TERM. So a tainted $TERM is considered to be safe if it contains only alphanumerics, underscores, dashes, and colons, and unsafe if it contains other characters (including whitespace).

New Opcode module and revised Safe module

A new Opcode module supports the creation, manipulation and application of opcode masks. The revised Safe module has a new API and is implemented using the new Opcode module. Please read the new Opcode and Safe documentation.

Embedding improvements

In older versions of Perl it was not possible to create more than one Perl interpreter instance inside a single process without leaking like a sieve and/or crashing. The bugs that caused this behavior have all been fixed. However, you still must take care when embedding Perl in a C program. See the updated perlembed manpage for tips on how to manage your interpreters.

Internal change: FileHandle class based on IO::* classes

File handles are now stored internally as type IO::Handle. The FileHandle module is still supported for backwards compatibility, but it is now merely a front end to the IO::* modules, specifically IO::Handle, IO::Seekable, and IO::File. We suggest, but do not require, that you use the IO::* modules in new code.

In harmony with this change, `*GLOB{FILEHANDLE}` is now just a backward-compatible synonym for `*GLOB{IO}`.

Internal change: PerlIO abstraction interface

It is now possible to build Perl with AT&T's sfio IO package instead of stdio. See *perlapio* for more details, and the *INSTALL* file for how to use it.

New and changed syntax

$coderef->(PARAMS)

A subroutine reference may now be suffixed with an arrow and a (possibly empty) parameter list. This syntax denotes a call of the referenced subroutine, with the given parameters (if any).

This new syntax follows the pattern of `$hashref->{FOO}` and `$aryref->[$foo]`: You may now write `&$subref($foo)` as `$subref->($foo)`. All these arrow terms may be chained; thus, `&{$table->{FOO}}($bar)` may now be written `$table->{FOO}->($bar)`.

New and changed builtin constants

__PACKAGE__

The current package name at compile time, or the undefined value if there is no current package (due to a `package;` directive). Like `__FILE__` and `__LINE__`, `__PACKAGE__` does *not* interpolate into strings.

New and changed builtin variables

$^E

Extended error message on some platforms. (Also known as $EXTENDED_OS_ERROR if you `use English`).

$^H

The current set of syntax checks enabled by `use strict`. See the documentation of `strict` for more details. Not actually new, but newly documented. Because it is intended for internal use by Perl core components, there is no `use English` long name for this variable.

$^M

By default, running out of memory it is not trappable. However, if compiled for this, Perl may use the contents of `$^M` as an emergency pool after die()ing with this message. Suppose that your Perl were compiled with -DPERL_EMERGENCY_SBRK and used Perl's malloc. Then

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

would allocate a 64K buffer for use when in emergency. See the *INSTALL* file for information on how to enable this option. As a disincentive to casual use of this advanced feature, there is no `use English` long name for this variable.

New and changed builtin functions

delete on slices

This now works. (e.g. `delete @ENV{'PATH', 'MANPATH'}`)

flock

is now supported on more platforms, prefers fcntl to lockf when emulating, and always flushes before (un)locking.

printf and sprintf

Perl now implements these functions itself; it doesn't use the C library function sprintf() any more, except for floating-point numbers, and even then only known flags are allowed. As a result, it is now possible to know which conversions and flags will work, and what they will do.

The new conversions in Perl's sprintf() are:

```
%i a synonym for %d
%p a pointer (the address of the Perl value, in hexadecimal)
%n special: *stores* the number of characters output so far
     into the next variable in the parameter list
```

The new flags that go between the % and the conversion are:

```
# prefix octal with "0", hex with "0x"
h interpret integer as C type "short" or "unsigned short"
V interpret integer as Perl's standard integer type
```

Also, where a number would appear in the flags, an asterisk ("*") may be used instead, in which case Perl uses the next item in the parameter list as the given number (that is, as the field width or precision). If a field width obtained through "*" is negative, it has the same effect as the '-' flag: left-justification.

See *"sprintf" in perlfunc* for a complete list of conversion and flags.

keys as an lvalue

As an lvalue, `keys` allows you to increase the number of hash buckets allocated for the given hash. This can gain you a measure of efficiency if you know the hash is going to get big. (This is similar to pre-extending an array by assigning a larger number to $#array.) If you say

```
keys %hash = 200;
```

then `%hash` will have at least 200 buckets allocated for it. These buckets will be retained even if you do `%hash = ()`; use `undef %hash` if you want to free the storage while `%hash` is still in scope. You can't shrink the number of buckets allocated for the hash using `keys` in this way (but you needn't worry about doing this by accident, as trying has no effect).

my() in Control Structures

You can now use my() (with or without the parentheses) in the control expressions of control structures such as:

```
while (defined(my $line = <>)) {
    $line = lc $line;
} continue {
    print $line;
}

if ((my $answer = <STDIN>) =~ /^y(es)?$/i) {
```

```
        user_agrees();
    } elsif ($answer =~ /^n(o)?$/i) {
        user_disagrees();
    } else {
        chomp $answer;
        die "`$answer' is neither `yes' nor `no'";
    }
```

Also, you can declare a foreach loop control variable as lexical by preceding it with the word "my". For example, in:

```
    foreach my $i (1, 2, 3) {
        some_function();
    }
```

$i is a lexical variable, and the scope of $i extends to the end of the loop, but not beyond it.

Note that you still cannot use my() on global punctuation variables such as $_ and the like.

pack() and unpack()

A new format 'w' represents a BER compressed integer (as defined in ASN.1). Its format is a sequence of one or more bytes, each of which provides seven bits of the total value, with the most significant first. Bit eight of each byte is set, except for the last byte, in which bit eight is clear.

If 'p' or 'P' are given undef as values, they now generate a NULL pointer.

Both pack() and unpack() now fail when their templates contain invalid types. (Invalid types used to be ignored.)

sysseek()

The new sysseek() operator is a variant of seek() that sets and gets the file's system read/write position, using the lseek(2) system call. It is the only reliable way to seek before using sysread() or syswrite(). Its return value is the new position, or the undefined value on failure.

use VERSION

If the first argument to `use` is a number, it is treated as a version number instead of a module name. If the version of the Perl interpreter is less than VERSION, then an error message is printed and Perl exits immediately. Because `use` occurs at compile time, this check happens immediately during the compilation process, unlike `require VERSION`, which waits until runtime for the check. This is often useful if you need to check the current Perl version before `use`ing library modules which have changed in incompatible ways from older versions of Perl. (We try not to do this more than we have to.)

use Module VERSION LIST

If the VERSION argument is present between Module and LIST, then the `use` will call the VERSION method in class Module with the given version as an argument. The default VERSION method, inherited from the UNIVERSAL class, croaks if the given version is larger than the value of the variable $Module::VERSION. (Note that there is not a comma after VERSION!)

This version-checking mechanism is similar to the one currently used in the Exporter module, but it is faster and can be used with modules that don't use the Exporter. It is the recommended method for new code.

prototype(FUNCTION)

Returns the prototype of a function as a string (or `undef` if the function has no prototype). FUNCTION is a reference to or the name of the function whose prototype you want to retrieve.

(Not actually new; just never documented before.)

srand

The default seed for `srand`, which used to be `time`, has been changed. Now it's a heady mix of difficult-to-predict system-dependent values, which should be sufficient for most everyday purposes.

Previous to version 5.004, calling `rand` without first calling `srand` would yield the same sequence of random numbers on most or all machines. Now, when perl sees that you're calling `rand` and haven't yet called `srand`, it calls `srand` with the default seed. You should still call `srand` manually if your code might ever be run on a pre-5.004 system, of course, or if you want a seed other than the default.

$_ as Default

Functions documented in the Camel to default to `$_` now in fact do, and all those that do are so documented in *perlfunc*.

`m//gc` does not reset search position on failure

The `m//g` match iteration construct has always reset its target string's search position (which is visible through the `pos` operator) when a match fails; as a result, the next `m//g` match after a failure starts again at the beginning of the string. With Perl 5.004, this reset may be disabled by adding the "c" (for "continue") modifier, i.e. `m//gc`. This feature, in conjunction with the `\G` zero-width assertion, makes it possible to chain matches together. See *perlop* and *perlre*.

`m//x` ignores whitespace before ?*+{}

The `m//x` construct has always been intended to ignore all unescaped whitespace. However, before Perl 5.004, whitespace had the effect of escaping repeat modifiers like "*" or "?"; for example, `/a *b/x` was (mis)interpreted as `/a\*b/x`. This bug has been fixed in 5.004.

nested `sub{}` closures work now

Prior to the 5.004 release, nested anonymous functions didn't work right. They do now.

formats work right on changing lexicals

Just like anonymous functions that contain lexical variables that change (like a lexical index variable for a `foreach` loop), formats now work properly. For example, this silently failed before (printed only zeros), but is fine now:

```
    my $i;
    foreach $i ( 1 .. 10 ) {
write;
    }
    format =
my i is @#
$i
    .
```

However, it still fails (without a warning) if the foreach is within a subroutine:

```
    my $i;
    sub foo {
      foreach $i ( 1 .. 10 ) {
write;
      }
    }
    foo;
    format =
my i is @#
$i
```

.

New builtin methods

The UNIVERSAL package automatically contains the following methods that are inherited by all other classes:

isa(CLASS)

isa returns *true* if its object is blessed into a subclass of CLASS

isa is also exportable and can be called as a sub with two arguments. This allows the ability to check what a reference points to. Example:

```
use UNIVERSAL qw(isa);

if(isa($ref, 'ARRAY')) {
   ...
}
```

can(METHOD)

can checks to see if its object has a method called METHOD, if it does then a reference to the sub is returned; if it does not then *undef* is returned.

VERSION([NEED])

VERSION returns the version number of the class (package). If the NEED argument is given then it will check that the current version (as defined by the $VERSION variable in the given package) not less than NEED; it will die if this is not the case. This method is normally called as a class method. This method is called automatically by the VERSION form of use.

```
use A 1.2 qw(some imported subs);
# implies:
A->VERSION(1.2);
```

NOTE: can directly uses Perl's internal code for method lookup, and isa uses a very similar method and caching strategy. This may cause strange effects if the Perl code dynamically changes @ISA in any package.

You may add other methods to the UNIVERSAL class via Perl or XS code. You do not need to use UNIVERSAL in order to make these methods available to your program. This is necessary only if you wish to have isa available as a plain subroutine in the current package.

TIEHANDLE now supported

See *perltie* for other kinds of tie()s.

TIEHANDLE classname, LIST

This is the constructor for the class. That means it is expected to return an object of some sort. The reference can be used to hold some internal information.

```
   sub TIEHANDLE {
print "<shout>\n";
my $i;
return bless \$i, shift;
   }
```

PRINT this, LIST

This method will be triggered every time the tied handle is printed to. Beyond its self reference it also expects the list that was passed to the print function.

```
    sub PRINT {
 $r = shift;
 $$r++;
 return print join( $, => map {uc} @_), $\;
    }
```

PRINTF this, LIST

This method will be triggered every time the tied handle is printed to with the `printf()` function. Beyond its self reference it also expects the format and list that was passed to the printf function.

```
    sub PRINTF {
        shift;
   my $fmt = shift;
        print sprintf($fmt, @_)."\n";
    }
```

READ this LIST

This method will be called when the handle is read from via the `read` or `sysread` functions.

```
    sub READ {
 $r = shift;
 my($buf,$len,$offset) = @_;
 print "READ called, \$buf=$buf, \$len=$len, \$offset=$offset";
    }
```

READLINE this

This method will be called when the handle is read from. The method should return undef when there is no more data.

```
    sub READLINE {
 $r = shift;
 return "PRINT called $$r times\n"
    }
```

GETC this

This method will be called when the `getc` function is called.

```
    sub GETC { print "Don't GETC, Get Perl"; return "a"; }
```

DESTROY this

As with the other types of ties, this method will be called when the tied handle is about to be destroyed. This is useful for debugging and possibly for cleaning up.

```
    sub DESTROY {
 print "</shout>\n";
    }
```

Malloc enhancements

If perl is compiled with the malloc included with the perl distribution (that is, if `perl -V:d_mymalloc` is 'define') then you can print memory statistics at runtime by running Perl thusly:

```
  env PERL_DEBUG_MSTATS=2 perl your_script_here
```

The value of 2 means to print statistics after compilation and on exit; with a value of 1, the statistics

are printed only on exit. (If you want the statistics at an arbitrary time, you'll need to install the optional module Devel::Peek.)

Three new compilation flags are recognized by malloc.c. (They have no effect if perl is compiled with system malloc().)

-DPERL_EMERGENCY_SBRK

If this macro is defined, running out of memory need not be a fatal error: a memory pool can allocated by assigning to the special variable `$^M`. See *$^M*.

-DPACK_MALLOC

Perl memory allocation is by bucket with sizes close to powers of two. Because of these malloc overhead may be big, especially for data of size exactly a power of two. If `PACK_MALLOC` is defined, perl uses a slightly different algorithm for small allocations (up to 64 bytes long), which makes it possible to have overhead down to 1 byte for allocations which are powers of two (and appear quite often).

Expected memory savings (with 8-byte alignment in `alignbytes`) is about 20% for typical Perl usage. Expected slowdown due to additional malloc overhead is in fractions of a percent (hard to measure, because of the effect of saved memory on speed).

-DTWO_POT_OPTIMIZE

Similarly to `PACK_MALLOC`, this macro improves allocations of data with size close to a power of two; but this works for big allocations (starting with 16K by default). Such allocations are typical for big hashes and special-purpose scripts, especially image processing.

On recent systems, the fact that perl requires 2M from system for 1M allocation will not affect speed of execution, since the tail of such a chunk is not going to be touched (and thus will not require real memory). However, it may result in a premature out-of-memory error. So if you will be manipulating very large blocks with sizes close to powers of two, it would be wise to define this macro.

Expected saving of memory is 0-100% (100% in applications which require most memory in such 2**n chunks); expected slowdown is negligible.

Miscellaneous efficiency enhancements

Functions that have an empty prototype and that do nothing but return a fixed value are now inlined (e.g. `sub PI () { 3.14159 }`).

Each unique hash key is only allocated once, no matter how many hashes have an entry with that key. So even if you have 100 copies of the same hash, the hash keys never have to be reallocated.

Support for More Operating Systems

Support for the following operating systems is new in Perl 5.004.

Win32

Perl 5.004 now includes support for building a "native" perl under Windows NT, using the Microsoft Visual C++ compiler (versions 2.0 and above) or the Borland C++ compiler (versions 5.02 and above). The resulting perl can be used under Windows 95 (if it is installed in the same directory locations as it got installed in Windows NT). This port includes support for perl extension building tools like *ExtUtils::MakeMaker* and *h2xs*, so that many extensions available on the Comprehensive Perl Archive Network (CPAN) can now be readily built under Windows NT. See http://www.perl.com/ for more information on CPAN and *README.win32* in the perl distribution for more details on how to get started with building this port.

There is also support for building perl under the Cygwin32 environment. Cygwin32 is a set of GNU tools that make it possible to compile and run many Unix programs under Windows NT by providing a mostly Unix-like interface for compilation and execution. See *README.cygwin32* in the perl distribution for more details on this port and how to obtain the Cygwin32 toolkit.

Plan 9

See *README.plan9* in the perl distribution.

QNX

See *README.qnx* in the perl distribution.

AmigaOS

See *README.amigaos* in the perl distribution.

Pragmata

Six new pragmatic modules exist:

use autouse MODULE => qw(sub1 sub2 sub3)

Defers `require MODULE` until someone calls one of the specified subroutines (which must be exported by MODULE). This pragma should be used with caution, and only when necessary.

use blib

use blib 'dir'

Looks for MakeMaker-like *'blib'* directory structure starting in *dir* (or current directory) and working back up to five levels of parent directories.

Intended for use on command line with **-M** option as a way of testing arbitrary scripts against an uninstalled version of a package.

use constant NAME => VALUE

Provides a convenient interface for creating compile-time constants, See *"Constant Functions" in perlsub*.

use locale

Tells the compiler to enable (or disable) the use of POSIX locales for builtin operations.

When `use locale` is in effect, the current LC_CTYPE locale is used for regular expressions and case mapping; LC_COLLATE for string ordering; and LC_NUMERIC for numeric formatting in printf and sprintf (but **not** in print). LC_NUMERIC is always used in write, since lexical scoping of formats is problematic at best.

Each `use locale` or `no locale` affects statements to the end of the enclosing BLOCK or, if not inside a BLOCK, to the end of the current file. Locales can be switched and queried with POSIX::setlocale().

See *perllocale* for more information.

use ops

Disable unsafe opcodes, or any named opcodes, when compiling Perl code.

use vmsish

Enable VMS-specific language features. Currently, there are three VMS-specific features available: 'status', which makes $? and `system` return genuine VMS status values instead of emulating POSIX; 'exit', which makes `exit` take a genuine VMS status value instead of assuming that `exit 1` is an error; and 'time', which makes all times relative to the local time zone, in the VMS tradition.

Modules

Required Updates

Though Perl 5.004 is compatible with almost all modules that work with Perl 5.003, there are a few exceptions:

```
Module   Required Version for Perl 5.004
------   ------------------------------
Filter   Filter-1.12
LWP      libwww-perl-5.08
Tk       Tk400.202 (-w makes noise)
```

Also, the majordomo mailing list program, version 1.94.1, doesn't work with Perl 5.004 (nor with perl 4), because it executes an invalid regular expression. This bug is fixed in majordomo version 1.94.2.

Installation directories

The *installperl* script now places the Perl source files for extensions in the architecture-specific library directory, which is where the shared libraries for extensions have always been. This change is intended to allow administrators to keep the Perl 5.004 library directory unchanged from a previous version, without running the risk of binary incompatibility between extensions' Perl source and shared libraries.

Module information summary

Brand new modules, arranged by topic rather than strictly alphabetically:

```
CGI.pm              Web server interface ("Common Gateway Interface")
CGI/Apache.pm       Support for Apache's Perl module
CGI/Carp.pm         Log server errors with helpful context
CGI/Fast.pm         Support for FastCGI (persistent server process)
CGI/Push.pm         Support for server push
CGI/Switch.pm       Simple interface for multiple server types

CPAN                Interface to Comprehensive Perl Archive Network
CPAN::FirstTime     Utility for creating CPAN configuration file
CPAN::Nox           Runs CPAN while avoiding compiled extensions

IO.pm               Top-level interface to IO::* classes
IO/File.pm          IO::File extension Perl module
IO/Handle.pm        IO::Handle extension Perl module
IO/Pipe.pm          IO::Pipe extension Perl module
IO/Seekable.pm      IO::Seekable extension Perl module
IO/Select.pm        IO::Select extension Perl module
IO/Socket.pm        IO::Socket extension Perl module

Opcode.pm           Disable named opcodes when compiling Perl code

ExtUtils/Embed.pm   Utilities for embedding Perl in C programs
ExtUtils/testlib.pm Fixes up @INC to use just-built extension

FindBin.pm          Find path of currently executing program

Class/Struct.pm     Declare struct-like datatypes as Perl classes
File/stat.pm        By-name interface to Perl's builtin stat
Net/hostent.pm      By-name interface to Perl's builtin gethost*
Net/netent.pm       By-name interface to Perl's builtin getnet*
Net/protoent.pm     By-name interface to Perl's builtin getproto*
Net/servent.pm      By-name interface to Perl's builtin getserv*
Time/gmtime.pm      By-name interface to Perl's builtin gmtime
Time/localtime.pm   By-name interface to Perl's builtin localtime
Time/tm.pm          Internal object for Time::{gm,local}time
```

```
    User/grent.pm        By-name interface to Perl's builtin getgr*
    User/pwent.pm        By-name interface to Perl's builtin getpw*

    Tie/RefHash.pm       Base class for tied hashes with references as keys

    UNIVERSAL.pm         Base class for *ALL* classes
```

Fcntl

New constants in the existing Fcntl modules are now supported, provided that your operating system happens to support them:

```
    F_GETOWN F_SETOWN
    O_ASYNC O_DEFER O_DSYNC O_FSYNC O_SYNC
    O_EXLOCK O_SHLOCK
```

These constants are intended for use with the Perl operators sysopen() and fcntl() and the basic database modules like SDBM_File. For the exact meaning of these and other Fcntl constants please refer to your operating system's documentation for fcntl() and open().

In addition, the Fcntl module now provides these constants for use with the Perl operator flock():

```
 LOCK_SH LOCK_EX LOCK_NB LOCK_UN
```

These constants are defined in all environments (because where there is no flock() system call, Perl emulates it). However, for historical reasons, these constants are not exported unless they are explicitly requested with the ":flock" tag (e.g. `use Fcntl ':flock'`).

IO

The IO module provides a simple mechanism to load all the IO modules at one go. Currently this includes:

```
     IO::Handle
     IO::Seekable
     IO::File
     IO::Pipe
     IO::Socket
```

For more information on any of these modules, please see its respective documentation.

Math::Complex

The Math::Complex module has been totally rewritten, and now supports more operations. These are overloaded:

```
     + - * / ** <=> neg ~ abs sqrt exp log sin cos atan2 "" (stringify)
```

And these functions are now exported:

```
    pi i Re Im arg
    log10 logn ln cbrt root
    tan
    csc sec cot
    asin acos atan
    acsc asec acot
    sinh cosh tanh
    csch sech coth
```

```
asinh acosh atanh
acsch asech acoth
cplx cplxe
```

Math::Trig

This new module provides a simpler interface to parts of Math::Complex for those who need trigonometric functions only for real numbers.

DB_File

There have been quite a few changes made to DB_File. Here are a few of the highlights:

- Fixed a handful of bugs.
- By public demand, added support for the standard hash function exists().
- Made it compatible with Berkeley DB 1.86.
- Made negative subscripts work with RECNO interface.
- Changed the default flags from O_RDWR to O_CREAT|O_RDWR and the default mode from 0640 to 0666.
- Made DB_File automatically import the open() constants (O_RDWR, O_CREAT etc.) from Fcntl, if available.
- Updated documentation.

Refer to the HISTORY section in DB_File.pm for a complete list of changes. Everything after DB_File 1.01 has been added since 5.003.

Net::Ping

Major rewrite - support added for both udp echo and real icmp pings.

Object-oriented overrides for builtin operators

Many of the Perl builtins returning lists now have object-oriented overrides. These are:

```
File::stat
Net::hostent
Net::netent
Net::protoent
Net::servent
Time::gmtime
Time::localtime
User::grent
User::pwent
```

For example, you can now say

```
use File::stat;
use User::pwent;
$his = (stat($filename)->st_uid == pwent($whoever)->pw_uid);
```

Utility Changes

pod2html

Sends converted HTML to standard output

The *pod2html* utility included with Perl 5.004 is entirely new. By default, it sends the converted HTML to its standard output, instead of writing it to a file like Perl 5.003's *pod2html* did. Use

the **--outfile=FILENAME** option to write to a file.

xsubpp

`void` XSUBs now default to returning nothing

Due to a documentation/implementation bug in previous versions of Perl, XSUBs with a return type of `void` have actually been returning one value. Usually that value was the GV for the XSUB, but sometimes it was some already freed or reused value, which would sometimes lead to program failure.

In Perl 5.004, if an XSUB is declared as returning `void`, it actually returns no value, i.e. an empty list (though there is a backward-compatibility exception; see below). If your XSUB really does return an SV, you should give it a return type of `SV *`.

For backward compatibility, *xsubpp* tries to guess whether a `void` XSUB is really `void` or if it wants to return an `SV *`. It does so by examining the text of the XSUB: if *xsubpp* finds what looks like an assignment to `ST(0)`, it assumes that the XSUB's return type is really `SV *`.

C Language API Changes

`gv_fetchmethod` and `perl_call_sv`

The `gv_fetchmethod` function finds a method for an object, just like in Perl 5.003. The GV it returns may be a method cache entry. However, in Perl 5.004, method cache entries are not visible to users; therefore, they can no longer be passed directly to `perl_call_sv`. Instead, you should use the `GvCV` macro on the GV to extract its CV, and pass the CV to `perl_call_sv`.

The most likely symptom of passing the result of `gv_fetchmethod` to `perl_call_sv` is Perl's producing an "Undefined subroutine called" error on the *second* call to a given method (since there is no cache on the first call).

`perl_eval_pv`

A new function handy for eval'ing strings of Perl code inside C code. This function returns the value from the eval statement, which can be used instead of fetching globals from the symbol table. See *perlguts*, *perlembed* and *perlcall* for details and examples.

Extended API for manipulating hashes

Internal handling of hash keys has changed. The old hashtable API is still fully supported, and will likely remain so. The additions to the API allow passing keys as `SV*`s, so that `tied` hashes can be given real scalars as keys rather than plain strings (nontied hashes still can only use strings as keys). New extensions must use the new hash access functions and macros if they wish to use `SV*` keys. These additions also make it feasible to manipulate `HE*`s (hash entries), which can be more efficient. See *perlguts* for details.

Documentation Changes

Many of the base and library pods were updated. These new pods are included in section 1:

perldelta

This document.

perlfaq

Frequently asked questions.

perllocale

Locale support (internationalization and localization).

perltoot

Tutorial on Perl OO programming.

perlapio

Perl internal IO abstraction interface.

perlmodlib

Perl module library and recommended practice for module creation. Extracted from *perlmod* (which is much smaller as a result).

perldebug

Although not new, this has been massively updated.

perlsec

Although not new, this has been massively updated.

New Diagnostics

Several new conditions will trigger warnings that were silent before. Some only affect certain platforms. The following new warnings and errors outline these. These messages are classified as follows (listed in increasing order of desperation):

```
(W) A warning (optional).
(D) A deprecation (optional).
(S) A severe warning (mandatory).
(F) A fatal error (trappable).
(P) An internal error you should never see (trappable).
(X) A very fatal error (nontrappable).
(A) An alien error message (not generated by Perl).
```

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

(W) A lexical variable has been redeclared in the same scope, 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.

%s argument is not a HASH element or slice

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

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

or a hash slice, such as

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

Allocation too large: %lx

(X) You can't allocate more than 64K on an MS-DOS machine.

Allocation too large

(F) You can't allocate more than 2^31+"small amount" bytes.

Applying %s to %s will act on scalar(%s)

(W) The pattern match (//), substitution (s///), and transliteration (tr///) operators work on scalar values. If you apply one of them to an array or a hash, it will convert the array or hash to a scalar value (the length of an array or the population info of a hash) and then work on that scalar value. This is probably not what you meant to do. See *"grep" in perlfunc* and *"map" in perlfunc* for alternatives.

Attempt to free nonexistent shared string

(P) Perl maintains a reference counted internal table of strings to optimize the storage and access of hash keys and other strings. This indicates someone tried to decrement the reference count of a string that can no longer be found in the table.

Attempt to use reference as lvalue in substr

(W) You supplied a reference as the first argument to substr() used as an lvalue, which is pretty strange. Perhaps you forgot to dereference it first. See *"substr" in perlfunc*.

Bareword "%s" refers to nonexistent package

(W) You used a qualified bareword of the form `Foo::`, but the compiler saw no other uses of that namespace before that point. Perhaps you need to predeclare a package?

Can't redefine active sort subroutine %s

(F) Perl optimizes the internal handling of sort subroutines and keeps pointers into them. You tried to redefine one such sort subroutine when it was currently active, which is not allowed. If you really want to do this, you should write `sort { &func } @x` instead of `sort func @x`.

Can't use bareword ("%s") as %s ref while "strict refs" in use

(F) Only hard references are allowed by "strict refs". Symbolic references are disallowed. See *perlref*.

Cannot resolve method `%s' overloading `%s' in package `%s'

(P) Internal error trying to resolve overloading specified by a method name (as opposed to a subroutine reference).

Constant subroutine %s redefined

(S) You redefined a subroutine which had previously been eligible for inlining. See *"Constant Functions" in perlsub* for commentary and workarounds.

Constant subroutine %s undefined

(S) You undefined a subroutine which had previously been eligible for inlining. See *"Constant Functions" in perlsub* for commentary and workarounds.

Copy method did not return a reference

(F) The method which overloads "=" is buggy. See *"Copy Constructor" in overload*.

Died

(F) You passed die() an empty string (the equivalent of `die ""`) or you called it with no args and both $@ and $_ were empty.

Exiting pseudo-block via %s

(W) You are exiting a rather special block construct (like a sort block or subroutine) by unconventional means, such as a goto, or a loop control statement. See *"sort" in perlfunc*.

Identifier too long

(F) Perl limits identifiers (names for variables, functions, etc.) to 252 characters for simple names, somewhat more for compound names (like `$A::B`). You've exceeded Perl's limits. Future versions of Perl are likely to eliminate these arbitrary limitations.

Illegal character %s (carriage return)

(F) A carriage return character was found in the input. This is an error, and not a warning, because carriage return characters can break multi-line strings, including here documents (e.g., `print <<EOF;`).

Illegal switch in PERL5OPT: %s

(X) The PERL5OPT environment variable may only be used to set the following switches:

-[DIMUdmw].

Integer overflow in hex number

(S) The literal hex number you have specified is too big for your architecture. On a 32-bit architecture the largest hex literal is 0xFFFFFFFF.

Integer overflow in octal number

(S) The literal octal number you have specified is too big for your architecture. On a 32-bit architecture the largest octal literal is 037777777777.

internal error: glob failed

(P) Something went wrong with the external program(s) used for `glob` and `<*.c>`. This may mean that your csh (C shell) is broken. If so, you should change all of the csh-related variables in config.sh: If you have tcsh, make the variables refer to it as if it were csh (e.g. `full_csh='/usr/bin/tcsh'`); otherwise, make them all empty (except that `d_csh` should be `'undef'`) so that Perl will think csh is missing. In either case, after editing config.sh, run `./Configure -S` and rebuild Perl.

Invalid conversion in %s: "%s"

(W) Perl does not understand the given format conversion. See *"sprintf" in perlfunc*.

Invalid type in pack: '%s'

(F) The given character is not a valid pack type. See *"pack" in perlfunc*.

Invalid type in unpack: '%s'

(F) The given character is not a valid unpack type. See *"unpack" in perlfunc*.

Name "%s::%s" used only once: possible typo

(W) Typographical errors often show up as unique variable names. If you had a good reason for having a unique name, then just mention it again somehow to suppress the message (the `use vars` pragma is provided for just this purpose).

Null picture in formline

(F) The first argument to formline must be a valid format picture specification. It was found to be empty, which probably means you supplied it an uninitialized value. See *perlform*.

Offset outside string

(F) You tried to do a read/write/send/recv operation with an offset pointing outside the buffer. This is difficult to imagine. The sole exception to this is that `sysread()`ing past the buffer will extend the buffer and zero pad the new area.

Out of memory!

(X|F) The malloc() function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request.

The request was judged to be small, so the possibility to trap it depends on the way Perl was compiled. By default it is not trappable. However, if compiled for this, Perl may use the contents of `$^M` as an emergency pool after die()ing with this message. In this case the error is trappable *once*.

Out of memory during request for %s

(F) The malloc() function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request. However, the request was judged large enough (compile-time default is 64K), so a possibility to shut down by trapping this error is granted.

panic: frexp

(P) The library function frexp() failed, making printf("%f") impossible.

Possible attempt to put comments in qw() list

(W) qw() lists contain items separated by whitespace; as with literal strings, comment characters are not ignored, but are instead treated as literal data. (You may have used different delimiters than the parentheses shown here; braces are also frequently used.)

You probably wrote something like this:

```
@list = qw(
    a # a comment
    b # another comment
);
```

when you should have written this:

```
@list = qw(
    a
    b
);
```

If you really want comments, build your list the old-fashioned way, with quotes and commas:

```
@list = (
    'a',    # a comment
    'b',    # another comment
);
```

Possible attempt to separate words with commas

(W) qw() lists contain items separated by whitespace; therefore commas aren't needed to separate the items. (You may have used different delimiters than the parentheses shown here; braces are also frequently used.)

You probably wrote something like this:

```
qw! a, b, c !;
```

which puts literal commas into some of the list items. Write it without commas if you don't want them to appear in your data:

```
qw! a b c !;
```

Scalar value @%s{%s} better written as $%s{%s}

(W) You've used a hash slice (indicated by @) to select a single element of a hash. Generally it's better to ask for a scalar value (indicated by $). The difference is that `$foo{&bar}` always behaves like a scalar, both when assigning to it and when evaluating its argument, while `@foo{&bar}` behaves like a list when you assign to it, and provides a list context to its subscript, which can do weird things if you're expecting only one subscript.

Stub found while resolving method `%s' overloading `%s' in %s

(P) Overloading resolution over @ISA tree may be broken by importing stubs. Stubs should never be implicitly created, but explicit calls to `can` may break this.

Too late for "**-T**" option

(X) The #! line (or local equivalent) in a Perl script contains the **-T** option, but Perl was not invoked with **-T** in its argument list. This is an error because, by the time Perl discovers a **-T** in a script, it's too late to properly taint everything from the environment. So Perl gives up.

untie attempted while %d inner references still exist

(W) A copy of the object returned from `tie` (or `tied`) was still valid when `untie` was called.

Unrecognized character %s

(F) The Perl parser has no idea what to do with the specified character in your Perl script (or eval). Perhaps you tried to run a compressed script, a binary program, or a directory as a Perl program.

Unsupported function fork

(F) Your version of executable does not support forking.

Note that under some systems, like OS/2, there may be different flavors of Perl executables, some of which may support fork, some not. Try changing the name you call Perl by to `perl_`, `perl__`, and so on.

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.

Value of %s can be "0"; test with defined()

(W) In a conditional expression, you used <HANDLE>, <*> (glob), `each()`, or `readdir()` as a boolean value. Each of these constructs can return a value of "0"; that would make the conditional expression false, which is probably not what you intended. When using these constructs in conditional expressions, test their values with the `defined` operator.

Variable "%s" may be unavailable

(W) An inner (nested) *anonymous* subroutine is inside a *named* subroutine, and outside that is another subroutine; and the anonymous (innermost) subroutine is referencing a lexical variable defined in the outermost subroutine. For example:

```
sub outermost { my $a; sub middle { sub { $a } } }
```

If the anonymous subroutine is called or referenced (directly or indirectly) from the outermost subroutine, it will share the variable as you would expect. But if the anonymous subroutine is called or referenced when the outermost subroutine is not active, it will see the value of the shared variable as it was before and during the *first* call to the outermost subroutine, which is probably not what you want.

In these circumstances, it is usually best to make the middle subroutine anonymous, using the `sub {}` syntax. Perl has specific support for shared variables in nested anonymous subroutines; a named subroutine in between interferes with this feature.

Variable "%s" will not stay shared

(W) An inner (nested) *named* subroutine is referencing a lexical variable defined in an outer subroutine.

When the inner subroutine is called, it will probably see the value of the outer subroutine's variable as it was before and during the *first* call to the outer subroutine; in this case, after the first call to the outer subroutine is complete, the inner and outer subroutines will no longer share a common value for the variable. In other words, the variable will no longer be shared.

Furthermore, if the outer subroutine is anonymous and references a lexical variable outside itself, then the outer and inner subroutines will *never* share the given variable.

This problem can usually be solved by making the inner subroutine anonymous, using the `sub {}` syntax. When inner anonymous subs that reference variables in outer subroutines are called or referenced, they are automatically rebound to the current values of such variables.

Warning: something's wrong

(W) You passed warn() an empty string (the equivalent of `warn ""`) or you called it with no args and $_ was empty.

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. Since 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.

Got an error from DosAllocMem

(P) An error peculiar to OS/2. Most probably you're using an obsolete version of Perl, and this should not happen anyway.

Malformed PERLLIB_PREFIX

(F) An error peculiar to OS/2. PERLLIB_PREFIX should be of the form

```
prefix1;prefix2
```

or

```
prefix1 prefix2
```

with nonempty prefix1 and prefix2. If `prefix1` is indeed a prefix of a builtin library search path, prefix2 is substituted. The error may appear if components are not found, or are too long. See "PERLLIB_PREFIX" in *README.os2*.

PERL_SH_DIR too long

(F) An error peculiar to OS/2. PERL_SH_DIR is the directory to find the `sh`-shell in. See "PERL_SH_DIR" in *README.os2*.

Process terminated by SIG%s

(W) This is a standard message issued by OS/2 applications, while *nix applications die in silence. It is considered a feature of the OS/2 port. One can easily disable this by appropriate sighandlers, see *"Signals" in perlipc*. See also "Process terminated by SIGTERM/SIGINT" in *README.os2*.

BUGS

If you find what you think is a bug, you might check the headers of recently posted articles in 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. Make sure you 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.com*> 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. This file has been significantly updated for 5.004, so even veteran users should look through it.

The *README* file for general stuff.

The *Copying* file for copyright information.

HISTORY

Constructed by Tom Christiansen, grabbing material with permission from innumerable contributors, with kibitzing by more than a few Perl porters.

Last update: Wed May 14 11:14:09 EDT 1997