NAME

perltrap - Perl traps for the unwary

DESCRIPTION

The biggest trap of all is forgetting to `use warnings` or use the **-w** switch; see *perllexwarn* and *perlrun*. The second biggest trap is not making your entire program runnable under `use strict`. The third biggest trap is not reading the list of changes in this version of Perl; see *perldelta*.

Awk Traps

Accustomed **awk** users should take special note of the following:

- A Perl program executes only once, not once for each input line. You can do an implicit loop with `-n` or `-p`.
- The English module, loaded via

  ```
  use English;
  ```

  allows you to refer to special variables (like `$/`) with names (like $RS), as though they were in **awk**; see *perlvar* for details.
- Semicolons are required after all simple statements in Perl (except at the end of a block). Newline is not a statement delimiter.
- Curly brackets are required on `if`s and `while`s.
- Variables begin with "$", "@" or "%" in Perl.
- Arrays index from 0. Likewise string positions in substr() and index().
- You have to decide whether your array has numeric or string indices.
- Hash values do not spring into existence upon mere reference.
- You have to decide whether you want to use string or numeric comparisons.
- Reading an input line does not split it for you. You get to split it to an array yourself. And the split() operator has different arguments than **awk**'s.
- The current input line is normally in $_, not $0. It generally does not have the newline stripped. ($0 is the name of the program executed.) See *perlvar*.
- $<*digit*> does not refer to fields--it refers to substrings matched by the last match pattern.
- The print() statement does not add field and record separators unless you set `$,` and `$\`. You can set $OFS and $ORS if you're using the English module.
- You must open your files before you print to them.
- The range operator is "..", not comma. The comma operator works as in C.
- The match operator is "=~", not "~". ("~" is the one's complement operator, as in C.)
- The exponentiation operator is "**", not "^". "^" is the XOR operator, as in C. (You know, one could get the feeling that **awk** is basically incompatible with C.)
- The concatenation operator is ".", not the null string. (Using the null string would render `/pat/ /pat/` unparsable, because the third slash would be interpreted as a division operator--the tokenizer is in fact slightly context sensitive for operators like "/", "?", and ">". And in fact, "." itself can be the beginning of a number.)
- The `next`, `exit`, and `continue` keywords work differently.

- The following variables work differently:

```
Awk Perl
ARGC scalar @ARGV (compare with $#ARGV)
ARGV[0] $0
FILENAME $ARGV
FNR $. - something
FS (whatever you like)
NF $#Fld, or some such
NR $.
OFMT $#
OFS $,
ORS $\
RLENGTH length($&)
RS $/
RSTART length($`)
SUBSEP $;
```

- You cannot set $RS to a pattern, only a string.
- When in doubt, run the **awk** construct through **a2p** and see what it gives you.

C/C++ Traps

Cerebral C and C++ programmers should take note of the following:

- Curly brackets are required on `if`'s and `while`'s.
- You must use `elsif` rather than `else if`.
- The `break` and `continue` keywords from C become in Perl `last` and `next`, respectively. Unlike in C, these do *not* work within a `do { } while` construct. See *"Loop Control" in perlsyn*.
- The switch statement is called `given/when` and only available in perl 5.10 or newer. See *"Switch statements" in perlsyn*.
- Variables begin with "$", "@" or "%" in Perl.
- Comments begin with "#", not "/*" or "//". Perl may interpret C/C++ comments as division operators, unterminated regular expressions or the defined-or operator.
- You can't take the address of anything, although a similar operator in Perl is the backslash, which creates a reference.
- ARGV must be capitalized. `$ARGV[0]` is C's `argv[1]`, and `argv[0]` ends up in `$0`.
- System calls such as link(), unlink(), rename(), etc. return nonzero for success, not 0. (system(), however, returns zero for success.)
- Signal handlers deal with signal names, not numbers. Use `kill -l` to find their names on your system.

Sed Traps

Seasoned **sed** programmers should take note of the following:

- A Perl program executes only once, not once for each input line. You can do an implicit loop with `-n` or `-p`.
- Backreferences in substitutions use "$" rather than "\".
- The pattern matching metacharacters "(", ")", and "|" do not have backslashes in front.

- The range operator is `...`, rather than comma.

Shell Traps

Sharp shell programmers should take note of the following:

- The backtick operator does variable interpolation without regard to the presence of single quotes in the command.
- The backtick operator does no translation of the return value, unlike **csh**.
- Shells (especially **csh**) do several levels of substitution on each command line. Perl does substitution in only certain constructs such as double quotes, backticks, angle brackets, and search patterns.
- Shells interpret scripts a little bit at a time. Perl compiles the entire program before executing it (except for `BEGIN` blocks, which execute at compile time).
- The arguments are available via @ARGV, not $1, $2, etc.
- The environment is not automatically made available as separate scalar variables.
- The shell's `test` uses "=", "!=", "<" etc for string comparisons and "-eq", "-ne", "-lt" etc for numeric comparisons. This is the reverse of Perl, which uses `eq`, `ne`, `lt` for string comparisons, and ==, `!=` < etc for numeric comparisons.

Perl Traps

Practicing Perl Programmers should take note of the following:

- Remember that many operations behave differently in a list context than they do in a scalar one. See *perldata* for details.
- Avoid barewords if you can, especially all lowercase ones. You can't tell by just looking at it whether a bareword is a function or a string. By using quotes on strings and parentheses on function calls, you won't ever get them confused.
- You cannot discern from mere inspection which builtins are unary operators (like chop() and chdir()) and which are list operators (like print() and unlink()). (Unless prototyped, user-defined subroutines can **only** be list operators, never unary ones.) See *perlop* and *perlsub*.
- People have a hard time remembering that some functions default to $_, or @ARGV, or whatever, but that others which you might expect to do not.
- The <FH> construct is not the name of the filehandle, it is a readline operation on that handle. The data read is assigned to $_ only if the file read is the sole condition in a while loop:

```
while (<FH>)      { }
while (defined($_ = <FH>)) { }..
<FH>;  # data discarded!
```

- Remember not to use = when you need =~; these two constructs are quite different:

```
$x =  /foo/;
$x =~ /foo/;
```

- The `do {}` construct isn't a real loop that you can use loop control on.
- Use `my()` for local variables whenever you can get away with it (but see *perlform* for where you can't). Using `local()` actually gives a local value to a global variable, which leaves you open to unforeseen side-effects of dynamic scoping.
- If you localize an exported variable in a module, its exported value will not change. The local

name becomes an alias to a new value but the external name is still an alias for the original.

Perl4 to Perl5 Traps

Practicing Perl4 Programmers should take note of the following Perl4-to-Perl5 specific traps.

They're crudely ordered according to the following list:

Discontinuance, Deprecation, and BugFix traps

Anything that's been fixed as a perl4 bug, removed as a perl4 feature or deprecated as a perl4 feature with the intent to encourage usage of some other perl5 feature.

Parsing Traps

Traps that appear to stem from the new parser.

Numerical Traps

Traps having to do with numerical or mathematical operators.

General data type traps

Traps involving perl standard data types.

Context Traps - scalar, list contexts

Traps related to context within lists, scalar statements/declarations.

Precedence Traps

Traps related to the precedence of parsing, evaluation, and execution of code.

General Regular Expression Traps using s///, etc.

Traps related to the use of pattern matching.

Subroutine, Signal, Sorting Traps

Traps related to the use of signals and signal handlers, general subroutines, and sorting, along with sorting subroutines.

OS Traps

OS-specific traps.

DBM Traps

Traps specific to the use of `dbmopen()`, and specific dbm implementations.

Unclassified Traps

Everything else.

If you find an example of a conversion trap that is not listed here, please submit it to <*perlbug@perl.org*> for inclusion. Also note that at least some of these can be caught with the `use warnings` pragma or the **-w** switch.

Discontinuance, Deprecation, and BugFix traps

Anything that has been discontinued, deprecated, or fixed as a bug from perl4.

* Symbols starting with "_" no longer forced into main

Symbols starting with "_" are no longer forced into package main, except for $_ itself (and @_, etc.).

```
package test;
$_legacy = 1;

package main;
```

```
print "\$_legacy is ",$_legacy,"\n";

# perl4 prints: $_legacy is 1
# perl5 prints: $_legacy is
```

* Double-colon valid package separator in variable name

Double-colon is now a valid package separator in a variable name. Thus these behave differently in perl4 vs. perl5, because the packages don't exist.

```
$a=1;$b=2;$c=3;$var=4;
print "$a::$b::$c ";
print "$var::abc::xyz\n";

# perl4 prints: 1::2::3 4::abc::xyz
# perl5 prints: 3
```

Given that `::` is now the preferred package delimiter, it is debatable whether this should be classed as a bug or not. (The older package delimiter, ' ,is used here)

```
$x = 10;
print "x=${'x}\n";

# perl4 prints: x=10
# perl5 prints: Can't find string terminator "'" anywhere before
EOF
```

You can avoid this problem, and remain compatible with perl4, if you always explicitly include the package name:

```
$x = 10;
print "x=${main'x}\n";
```

Also see precedence traps, for parsing `$:`.

* 2nd and 3rd args to `splice()` are now in scalar context

The second and third arguments of `splice()` are now evaluated in scalar context (as the Camel says) rather than list context.

```
sub sub1{return(0,2) }          # return a 2-element list
sub sub2{ return(1,2,3)}        # return a 3-element list
@a1 = ("a","b","c","d","e");
@a2 = splice(@a1,&sub1,&sub2);
print join(' ',@a2),"\n";

# perl4 prints: a b
# perl5 prints: c d e
```

* Can't do `goto` into a block that is optimized away

You can't do a `goto` into a block that is optimized away. Darn.

```
goto marker1;

for(1){
marker1:
    print "Here I is!\n";
}

# perl4 prints: Here I is!
```

```
    # perl5 errors: Can't "goto" into the middle of a foreach loop
```

* Can't use whitespace as variable name or quote delimiter

  It is no longer syntactically legal to use whitespace as the name of a variable, or as a delimiter for any kind of quote construct. Double darn.

```
    $a = ("foo bar");
    $b = q baz ;
    print "a is $a, b is $b\n";

    # perl4 prints: a is foo bar, b is baz
    # perl5 errors: Bareword found where operator expected
```

* `while/if BLOCK BLOCK` gone

  The archaic while/if BLOCK BLOCK syntax is no longer supported.

```
    if { 1 } {
        print "True!";
    }
    else {
        print "False!";
    }

    # perl4 prints: True!
    # perl5 errors: syntax error at test.pl line 1, near "if {"
```

* `**` binds tighter than unary minus

  The `**` operator now binds more tightly than unary minus. It was documented to work this way before, but didn't.

```
    print -4**2,"\n";

    # perl4 prints: 16
    # perl5 prints: -16
```

* `foreach` changed when iterating over a list

  The meaning of `foreach{}` has changed slightly when it is iterating over a list which is not an array. This used to assign the list to a temporary array, but no longer does so (for efficiency). This means that you'll now be iterating over the actual values, not over copies of the values. Modifications to the loop variable can change the original values.

```
    @list = ('ab','abc','bcd','def');
    foreach $var (grep(/ab/,@list)){
        $var = 1;
    }
    print (join(':',@list));

    # perl4 prints: ab:abc:bcd:def
    # perl5 prints: 1:1:bcd:def
```

  To retain Perl4 semantics you need to assign your list explicitly to a temporary array and then iterate over that. For example, you might need to change

```
    foreach $var (grep(/ab/,@list)){
```

  to

```
foreach $var (@tmp = grep(/ab/,@list)){
```

Otherwise changing $var will clobber the values of @list. (This most often happens when you use $_ for the loop variable, and call subroutines in the loop that don't properly localize $_.)

* `split` with no args behavior changed

`split` with no arguments now behaves like `split ' '` (which doesn't return an initial null field if $_ starts with whitespace), it used to behave like `split /\s+/` (which does).

```
$_ = ' hi mom';
print join(':', split);

# perl4 prints: :hi:mom
# perl5 prints: hi:mom
```

* **-e** behavior fixed

Perl 4 would ignore any text which was attached to an **-e** switch, always taking the code snippet from the following arg. Additionally, it would silently accept an **-e** switch without a following arg. Both of these behaviors have been fixed.

```
perl -e'print "attached to -e"' 'print "separate arg"'

# perl4 prints: separate arg
# perl5 prints: attached to -e

perl -e

# perl4 prints:
# perl5 dies: No code specified for -e.
```

* `push` returns number of elements in resulting list

In Perl 4 the return value of `push` was undocumented, but it was actually the last value being pushed onto the target list. In Perl 5 the return value of `push` is documented, but has changed, it is the number of elements in the resulting list.

```
@x = ('existing');
print push(@x, 'first new', 'second new');

# perl4 prints: second new
# perl5 prints: 3
```

* Some error messages differ

Some error messages will be different.

* `split()` honors subroutine args

In Perl 4, if in list context the delimiters to the first argument of `split()` were ??, the result would be placed in @_ as well as being returned. Perl 5 has more respect for your subroutine arguments.

* Bugs removed

Some bugs may have been inadvertently removed. :-)

Parsing Traps

Perl4-to-Perl5 traps from having to do with parsing.

* Space between . and = triggers syntax error

Note the space between . and =

```
    $string . = "more string";
    print $string;

    # perl4 prints: more string
    # perl5 prints: syntax error at - line 1, near ". ="
```

* Better parsing in perl 5

Better parsing in perl 5

```
    sub foo {}
    &foo
    print("hello, world\n");

    # perl4 prints: hello, world
    # perl5 prints: syntax error
```

* Function parsing

"if it looks like a function, it is a function" rule.

```
  print
    ($foo == 1) ? "is one\n" : "is zero\n";

    # perl4 prints: is zero
    # perl5 warns: "Useless use of a constant in void context" if
using -w
```

* String interpolation of `$#array` differs

String interpolation of the `$#array` construct differs when braces are to used around the name.

```
    @a = (1..3);
    print "${#a}";

    # perl4 prints: 2
    # perl5 fails with syntax error

    @a = (1..3);
    print "$#{a}";

    # perl4 prints: {a}
    # perl5 prints: 2
```

* Perl guesses on `map`, `grep` followed by `{` if it starts BLOCK or hash ref

When perl sees `map {` (or `grep {`), it has to guess whether the `{` starts a BLOCK or a hash reference. If it guesses wrong, it will report a syntax error near the `}` and the missing (or unexpected) comma.

Use unary + before `{` on a hash reference, and unary + applied to the first thing in a BLOCK (after `{`), for perl to guess right all the time. (See *"map" in perlfunc*.)

Numerical Traps

Perl4-to-Perl5 traps having to do with numerical operators, operands, or output from same.

* Formatted output and significant digits

Formatted output and significant digits. In general, Perl 5 tries to be more precise. For

example, on a Solaris Sparc:

```
print 7.373504 - 0, "\n";
printf "%20.18f\n", 7.373504 - 0;

# Perl4 prints:
7.3750399999999996141
7.375039999999999614

# Perl5 prints:
7.373504
7.3735039999999999614
```

Notice how the first result looks better in Perl 5.

Your results may vary, since your floating point formatting routines and even floating point format may be slightly different.

* Auto-increment operator over signed int limit deleted

This specific item has been deleted. It demonstrated how the auto-increment operator would not catch when a number went over the signed int limit. Fixed in version 5.003_04. But always be wary when using large integers. If in doubt:

```
use Math::BigInt;
```

* Assignment of return values from numeric equality tests doesn't work

Assignment of return values from numeric equality tests does not work in perl5 when the test evaluates to false (0). Logical tests now return a null, instead of 0

```
$p = ($test == 1);
print $p,"\n";

# perl4 prints: 0
# perl5 prints:
```

Also see *General Regular Expression Traps using s///, etc.* for another example of this new feature...

* Bitwise string ops

When bitwise operators which can operate upon either numbers or strings (`& | ^ ~`) are given only strings as arguments, perl4 would treat the operands as bitstrings so long as the program contained a call to the `vec()` function. perl5 treats the string operands as bitstrings. (See *"Bitwise String Operators" in perlop* for more details.)

```
$fred = "10";
$barney = "12";
$betty = $fred & $barney;
print "$betty\n";
# Uncomment the next line to change perl4's behavior
# ($dummy) = vec("dummy", 0, 0);

# Perl4 prints:
8

# Perl5 prints:
10

# If vec() is used anywhere in the program, both print:
10
```

General data type traps

Perl4-to-Perl5 traps involving most data-types, and their usage within certain expressions and/or context.

* Negative array subscripts now count from the end of array

  Negative array subscripts now count from the end of the array.

```
    @a = (1, 2, 3, 4, 5);
    print "The third element of the array is $a[3] also expressed as
 $a[-2] \n";

    # perl4 prints: The third element of the array is 4 also
expressed as
    # perl5 prints: The third element of the array is 4 also
expressed as 4
```

* Setting `$#array` lower now discards array elements

  Setting `$#array` lower now discards array elements, and makes them impossible to recover.

```
    @a = (a,b,c,d,e);
    print "Before: ",join('',@a);
    $#a =1;
    print ", After: ",join('',@a);
    $#a =3;
    print ", Recovered: ",join('',@a),"\n";

    # perl4 prints: Before: abcde, After: ab, Recovered: abcd
    # perl5 prints: Before: abcde, After: ab, Recovered: ab
```

* Hashes get defined before use

  Hashes get defined before use

```
    local($s,@a,%h);
    die "scalar \$s defined" if defined($s);
    die "array \@a defined" if defined(@a);
    die "hash \%h defined" if defined(%h);

    # perl4 prints:
    # perl5 dies: hash %h defined
```

  Perl will now generate a warning when it sees defined(@a) and defined(%h).

* Glob assignment from localized variable to variable

  glob assignment from variable to variable will fail if the assigned variable is localized subsequent to the assignment

```
    @a = ("This is Perl 4");
    *b = *a;
    local(@a);
    print @b,"\n";

    # perl4 prints: This is Perl 4
    # perl5 prints:
```

* Assigning `undef` to glob

Assigning `undef` to a glob has no effect in Perl 5. In Perl 4 it undefines the associated scalar (but may have other side effects including SEGVs). Perl 5 will also warn if `undef` is assigned to a typeglob. (Note that assigning `undef` to a typeglob is different than calling the `undef` function on a typeglob (`undef *foo`), which has quite a few effects.

```
$foo = "bar";
*foo = undef;
print $foo;

# perl4 prints:
# perl4 warns: "Use of uninitialized variable" if using -w
# perl5 prints: bar
# perl5 warns: "Undefined value assigned to typeglob" if using
-w
```

* Changes in unary negation (of strings)

Changes in unary negation (of strings) This change effects both the return value and what it does to auto(magic)increment.

```
$x = "aaa";
print ++$x," : ";
print -$x," : ";
print ++$x,"\n";

# perl4 prints: aab : -0 : 1
# perl5 prints: aab : -aab : aac
```

* Modifying of constants prohibited

perl 4 lets you modify constants:

```
$foo = "x";
&mod($foo);
for ($x = 0; $x < 3; $x++) {
    &mod("a");
}
sub mod {
    print "before: $_[0]";
    $_[0] = "m";
    print "  after: $_[0]\n";
}

# perl4:
# before: x  after: m
# before: a  after: m
# before: m  after: m
# before: m  after: m

# Perl5:
# before: x  after: m
# Modification of a read-only value attempted at foo.pl line 12.
# before: a
```

* `defined $var` behavior changed

The behavior is slightly different for:

```
print "$x", defined $x
```

```
    # perl 4: 1
    # perl 5: <no output, $x is not called into existence>
```

* Variable Suicide

Variable suicide behavior is more consistent under Perl 5. Perl5 exhibits the same behavior for hashes and scalars, that perl4 exhibits for only scalars.

```
    $aGlobal{ "aKey" } = "global value";
    print "MAIN:", $aGlobal{"aKey"}, "\n";
    $GlobalLevel = 0;
    &test( *aGlobal );

    sub test {
        local( *theArgument ) = @_;
        local( %aNewLocal ); # perl 4 != 5.001l,m
        $aNewLocal{"aKey"} = "this should never appear";
        print "SUB: ", $theArgument{"aKey"}, "\n";
        $aNewLocal{"aKey"} = "level $GlobalLevel";   # what should
print
        $GlobalLevel++;
        if( $GlobalLevel<4 ) {
            &test( *aNewLocal );
        }
    }

    # Perl4:
    # MAIN:global value
    # SUB: global value
    # SUB: level 0
    # SUB: level 1
    # SUB: level 2

    # Perl5:
    # MAIN:global value
    # SUB: global value
    # SUB: this should never appear
    # SUB: this should never appear
    # SUB: this should never appear
```

Context Traps - scalar, list contexts

* Elements of argument lists for formats evaluated in list context

The elements of argument lists for formats are now evaluated in list context. This means you can interpolate list values now.

```
    @fmt = ("foo","bar","baz");
    format STDOUT=
    @<<<<< @||||| @>>>>>
    @fmt;
    .
    write;

    # perl4 errors:  Please use commas to separate fields in file
    # perl5 prints: foo     bar      baz
```

* `caller()` returns false value in scalar context if no caller present

The `caller()` function now returns a false value in a scalar context if there is no caller. This lets library files determine if they're being required.

```
caller() ? (print "You rang?\n") : (print "Got a 0\n");

# perl4 errors: There is no caller
# perl5 prints: Got a 0
```

* Comma operator in scalar context gives scalar context to args

The comma operator in a scalar context is now guaranteed to give a scalar context to its last argument. It gives scalar or void context to any preceding arguments, depending on circumstances.

```
@y= ('a','b','c');
$x = (1, 2, @y);
print "x = $x\n";

# Perl4 prints:  x = c   # Interpolates array @y into the list
# Perl5 prints:  x = 3   # Evaluates array @y in scalar context
```

* `sprintf()` prototyped as `($;@)`

`sprintf()` is prototyped as ($;@), so its first argument is given scalar context. Thus, if passed an array, it will probably not do what you want, unlike Perl 4:

```
@z = ('%s%s', 'foo', 'bar');
$x = sprintf(@z);
print $x;

# perl4 prints: foobar
# perl5 prints: 3
```

`printf()` works the same as it did in Perl 4, though:

```
@z = ('%s%s', 'foo', 'bar');
printf STDOUT (@z);

# perl4 prints: foobar
# perl5 prints: foobar
```

Precedence Traps

Perl4-to-Perl5 traps involving precedence order.

Perl 4 has almost the same precedence rules as Perl 5 for the operators that they both have. Perl 4 however, seems to have had some inconsistencies that made the behavior differ from what was documented.

* LHS vs. RHS of any assignment operator

LHS vs. RHS of any assignment operator. LHS is evaluated first in perl4, second in perl5; this can affect the relationship between side-effects in sub-expressions.

```
@arr = ( 'left', 'right' );
$a{shift @arr} = shift @arr;
print join( ' ', keys %a );

# perl4 prints: left
# perl5 prints: right
```

* Semantic errors introduced due to precedence

These are now semantic errors because of precedence:

```
@list = (1,2,3,4,5);
%map = ("a",1,"b",2,"c",3,"d",4);
$n = shift @list + 2;   # first item in list plus 2
print "n is $n, ";
$m = keys %map + 2;     # number of items in hash plus 2
print "m is $m\n";

# perl4 prints: n is 3, m is 6
# perl5 errors and fails to compile
```

* Precedence of assignment operators same as the precedence of assignment

The precedence of assignment operators is now the same as the precedence of assignment. Perl 4 mistakenly gave them the precedence of the associated operator. So you now must parenthesize them in expressions like

```
/foo/ ? ($a += 2) : ($a -= 2);
```

Otherwise

```
/foo/ ? $a += 2 : $a -= 2
```

would be erroneously parsed as

```
(/foo/ ? $a += 2 : $a) -= 2;
```

On the other hand,

```
$a += /foo/ ? 1 : 2;
```

now works as a C programmer would expect.

* `open` requires parentheses around filehandle

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

is now incorrect. You need parentheses around the filehandle. Otherwise, perl5 leaves the statement as its default precedence:

```
open(FOO || die);

# perl4 opens or dies
# perl5 opens FOO, dying only if 'FOO' is false, i.e. never
```

* `$:` precedence over `$::` gone

perl4 gives the special variable, `$:` precedence, where perl5 treats `$::` as main `package`

```
$a = "x"; print "$::a";

# perl 4 prints: -:a
# perl 5 prints: x
```

* Precedence of file test operators documented

perl4 had buggy precedence for the file test operators vis-a-vis the assignment operators. Thus, although the precedence table for perl4 leads one to believe `-e $foo .= "q"` should parse as `((-e $foo) .= "q")`, it actually parses as `(-e ($foo .= "q"))`. In perl5, the precedence is as documented.

```
    -e $foo .= "q"

    # perl4 prints: no output
    # perl5 prints: Can't modify -e in concatenation
```

* `keys`, `each`, `values` are regular named unary operators

  In perl4, keys(), each() and values() were special high-precedence operators that operated on a single hash, but in perl5, they are regular named unary operators. As documented, named unary operators have lower precedence than the arithmetic and concatenation operators `+ - .`, but the perl4 variants of these operators actually bind tighter than `+ - ..` Thus, for:

```
    %foo = 1..10;
    print keys %foo - 1

    # perl4 prints: 4
    # perl5 prints: Type of arg 1 to keys must be hash (not
subtraction)
```

  The perl4 behavior was probably more useful, if less consistent.

General Regular Expression Traps using s///, etc.

All types of RE traps.

* `s'$lhs'$rhs'` interpolates on either side

  `s'$lhs'$rhs'` now does no interpolation on either side. It used to interpolate $lhs but not $rhs. (And still does not match a literal '$' in string)

```
    $a=1;$b=2;
    $string = '1 2 $a $b';
    $string =~ s'$a'$b';
    print $string,"\n";

    # perl4 prints: $b 2 $a $b
    # perl5 prints: 1 2 $a $b
```

* `m//g` attaches its state to the searched string

  `m//g` now attaches its state to the searched string rather than the regular expression. (Once the scope of a block is left for the sub, the state of the searched string is lost)

```
    $_ = "abababab";
    while(m/ab/g){
        &doit("blah");
    }
    sub doit{local($_) = shift; print "Got $_ "}

    # perl4 prints: Got blah Got blah Got blah Got blah
    # perl5 prints: infinite loop blah...
```

* `m//o` used within an anonymous sub

  Currently, if you use the `m//o` qualifier on a regular expression within an anonymous sub, *all* closures generated from that anonymous sub will use the regular expression as it was compiled when it was used the very first time in any such closure. For instance, if you say

```
    sub build_match {
        my($left,$right) = @_;
        return sub { $_[0] =~ /$left stuff $right/o; };
```

```
    }
    $good = build_match('foo','bar');
    $bad = build_match('baz','blarch');
    print $good->('foo stuff bar') ? "ok\n" : "not ok\n";
    print $bad->('baz stuff blarch') ? "ok\n" : "not ok\n";
    print $bad->('foo stuff bar') ? "not ok\n" : "ok\n";
```

For most builds of Perl5, this will print: ok not ok not ok

build_match() will always return a sub which matches the contents of $left and $right as they were the *first* time that build_match() was called, not as they are in the current call.

* $+ isn't set to whole match

If no parentheses are used in a match, Perl4 sets $+ to the whole match, just like $&. Perl5 does not.

```
    "abcdef" =~ /b.*e/;
    print "\$+ = $+\n";

    # perl4 prints: bcde
    # perl5 prints:
```

* Substitution now returns null string if it fails

substitution now returns the null string if it fails

```
    $string = "test";
    $value = ($string =~ s/foo//);
    print $value, "\n";

    # perl4 prints: 0
    # perl5 prints:
```

Also see *Numerical Traps* for another example of this new feature.

* `s`lhs`rhs`` is now a normal substitution

`s`lhs`rhs`` (using backticks) is now a normal substitution, with no backtick expansion

```
    $string = "";
    $string =~ s`^`hostname`;
    print $string, "\n";

    # perl4 prints: <the local hostname>
    # perl5 prints: hostname
```

* Stricter parsing of variables in regular expressions

Stricter parsing of variables used in regular expressions

```
    s/^([^$grpc]*$grpc[$opt$plus$rep]?)//o;

    # perl4: compiles w/o error
    # perl5: with Scalar found where operator expected ..., near
"$opt$plus"
```

an added component of this example, apparently from the same script, is the actual value of the s'd string after the substitution. `[$opt]` is a character class in perl4 and an array subscript in perl5

```
    $grpc = 'a';
    $opt  = 'r';
```

```
    $_ = 'bar';
    s/^([^$grpc]*$grpc[$opt]?)/foo/;
    print;

    # perl4 prints: foo
    # perl5 prints: foobar
```

* m?x? matches only once

  Under perl5, m?x? matches only once, like ?x?. Under perl4, it matched repeatedly, like /x/ or m!x!.

```
    $test = "once";
    sub match { $test =~ m?once?; }
    &match();
    if( &match() ) {
        # m?x? matches more then once
        print "perl4\n";
    } else {
        # m?x? matches only once
        print "perl5\n";
    }

    # perl4 prints: perl4
    # perl5 prints: perl5
```

* Failed matches don't reset the match variables

  Unlike in Ruby, failed matches in Perl do not reset the match variables ($1, $2, ..., $`, ...).

Subroutine, Signal, Sorting Traps

The general group of Perl4-to-Perl5 traps having to do with Signals, Sorting, and their related subroutines, as well as general subroutine traps. Includes some OS-Specific traps.

* Barewords that used to look like strings look like subroutine calls

  Barewords that used to look like strings to Perl will now look like subroutine calls if a subroutine by that name is defined before the compiler sees them.

```
    sub SeeYa { warn"Hasta la vista, baby!" }
    $SIG{'TERM'} = SeeYa;
    print "SIGTERM is now $SIG{'TERM'}\n";

    # perl4 prints: SIGTERM is now main'SeeYa
    # perl5 prints: SIGTERM is now main::1 (and warns "Hasta la
vista, baby!")
```

  Use **-w** to catch this one

* Reverse is no longer allowed as the name of a sort subroutine

  reverse is no longer allowed as the name of a sort subroutine.

```
    sub reverse{ print "yup "; $a <=> $b }
    print sort reverse (2,1,3);

    # perl4 prints: yup yup 123
    # perl5 prints: 123
    # perl5 warns (if using -w): Ambiguous call resolved as
CORE::reverse()
```

* `warn()` won't let you specify a filehandle.

Although it _always_ printed to STDERR, warn() would let you specify a filehandle in perl4. With perl5 it does not.

```
warn STDERR "Foo!";

# perl4 prints: Foo!
# perl5 prints: String found where operator expected
```

OS Traps

* SysV resets signal handler correctly

Under HPUX, and some other SysV OSes, one had to reset any signal handler, within the signal handler function, each time a signal was handled with perl4. With perl5, the reset is now done correctly. Any code relying on the handler _not_ being reset will have to be reworked.

Since version 5.002, Perl uses sigaction() under SysV.

```
sub gotit {
    print "Got @_... ";
}
$SIG{'INT'} = 'gotit';

$| = 1;
$pid = fork;
if ($pid) {
    kill('INT', $pid);
    sleep(1);
    kill('INT', $pid);
} else {
    while (1) {sleep(10);}
}

# perl4 (HPUX) prints: Got INT...
# perl5 (HPUX) prints: Got INT... Got INT...
```

* SysV `seek()` appends correctly

Under SysV OSes, `seek()` on a file opened to append >> now does the right thing w.r.t. the fopen() manpage. e.g., - When a file is opened for append, it is impossible to overwrite information already in the file.

```
open(TEST,">>seek.test");
$start = tell TEST;
foreach(1 .. 9){
    print TEST "$_ ";
}
$end = tell TEST;
seek(TEST,$start,0);
print TEST "18 characters here";

# perl4 (solaris) seek.test has: 18 characters here
# perl5 (solaris) seek.test has: 1 2 3 4 5 6 7 8 9 18 characters
 here
```

Interpolation Traps

Perl4-to-Perl5 traps having to do with how things get interpolated within certain expressions, statements, contexts, or whatever.

* @ always interpolates an array in double-quotish strings

@ now always interpolates an array in double-quotish strings.

```
    print "To: someone@somewhere.com\n";

    # perl4 prints: To:someone@somewhere.com
    # perl < 5.6.1, error : In string, @somewhere now must be
written as \@somewhere
    # perl >= 5.6.1, warning : Possible unintended interpolation of
@somewhere in string
```

* Double-quoted strings may no longer end with an unescaped $

Double-quoted strings may no longer end with an unescaped $.

```
    $foo = "foo$";
    print "foo is $foo\n";

    # perl4 prints: foo is foo$
    # perl5 errors: Final $ should be \$ or $name
```

Note: perl5 DOES NOT error on the terminating @ in $bar

* Arbitrary expressions are evaluated inside braces within double quotes

Perl now sometimes evaluates arbitrary expressions inside braces that occur within double quotes (usually when the opening brace is preceded by $ or @).

```
    @www = "buz";
    $foo = "foo";
    $bar = "bar";
    sub foo { return "bar" };
    print "|@{w.w.w}|${main'foo}|";

    # perl4 prints: |@{w.w.w}|foo|
    # perl5 prints: |buz|bar|
```

Note that you can `use strict;` to ward off such trappiness under perl5.

* $$x now tries to dereference $x

The construct "this is $$x" used to interpolate the pid at that point, but now tries to dereference $x. $$ by itself still works fine, however.

```
    $s = "a reference";
    $x = *s;
    print "this is $$x\n";

    # perl4 prints: this is XXXx   (XXX is the current pid)
    # perl5 prints: this is a reference
```

* Creation of hashes on the fly with `eval "EXPR"` requires protection

Creation of hashes on the fly with `eval "EXPR"` now requires either both $'s to be protected in the specification of the hash name, or both curlies to be protected. If both curlies are protected, the result will be compatible with perl4 and perl5. This is a very common practice, and should be changed to use the block form of `eval{}` if possible.

```
    $hashname = "foobar";
    $key = "baz";
    $value = 1234;
    eval "\$$hashname{'$key'} = q|$value|";
    (defined($foobar{'baz'})) ?  (print "Yup") : (print "Nope");

    # perl4 prints: Yup
    # perl5 prints: Nope
```

Changing

```
    eval "\$$hashname{'$key'} = q|$value|";
```

to

```
    eval "\$\$hashname{'$key'} = q|$value|";
```

causes the following result:

```
    # perl4 prints: Nope
    # perl5 prints: Yup
```

or, changing to

```
    eval "\$$hashname\{'$key'\} = q|$value|";
```

causes the following result:

```
    # perl4 prints: Yup
    # perl5 prints: Yup
    # and is compatible for both versions
```

* Bugs in earlier perl versions

perl4 programs which unconsciously rely on the bugs in earlier perl versions.

```
    perl -e '$bar=q/not/; print "This is $foo{$bar} perl5"'

    # perl4 prints: This is not perl5
    # perl5 prints: This is perl5
```

* Array and hash brackets during interpolation

You also have to be careful about array and hash brackets during interpolation.

```
    print "$foo["

    perl 4 prints: [
    perl 5 prints: syntax error

    print "$foo{"

    perl 4 prints: {
    perl 5 prints: syntax error
```

Perl 5 is expecting to find an index or key name following the respective brackets, as well as an ending bracket of the appropriate type. In order to mimic the behavior of Perl 4, you must escape the bracket like so.

```
    print "$foo\[";
    print "$foo\{";
```

* Interpolation of `\$$foo{bar}`

Similarly, watch out for: `\$$foo{bar}`

```
$foo = "baz";
print "\$$foo{bar}\n";

# perl4 prints: $baz{bar}
# perl5 prints: $
```

Perl 5 is looking for `$foo{bar}` which doesn't exist, but perl 4 is happy just to expand $foo to "baz" by itself. Watch out for this especially in `eval`'s.

* `qq()` string passed to `eval` will not find string terminator

`qq()` string passed to `eval`

```
eval qq(
    foreach \$y (keys %\$x\) {
        \$count++;
    }
);

# perl4 runs this ok
# perl5 prints: Can't find string terminator ")"
```

DBM Traps

General DBM traps.

* Perl5 must have been linked with same dbm/ndbm as the default for `dbmopen()`

Existing dbm databases created under perl4 (or any other dbm/ndbm tool) may cause the same script, run under perl5, to fail. The build of perl5 must have been linked with the same dbm/ndbm as the default for `dbmopen()` to function properly without `tie`'ing to an extension dbm implementation.

```
dbmopen (%dbm, "file", undef);
print "ok\n";

# perl4 prints: ok
# perl5 prints: ok (IFF linked with -ldbm or -lndbm)
```

* DBM exceeding limit on the key/value size will cause perl5 to exit immediately

Existing dbm databases created under perl4 (or any other dbm/ndbm tool) may cause the same script, run under perl5, to fail. The error generated when exceeding the limit on the key/value size will cause perl5 to exit immediately.

```
dbmopen(DB, "testdb",0600) || die "couldn't open db! $!";
$DB{'trap'} = "x" x 1024;  # value too large for most dbm/ndbm
print "YUP\n";

# perl4 prints:
dbm store returned -1, errno 28, key "trap" at - line 3.
YUP

# perl5 prints:
dbm store returned -1, errno 28, key "trap" at - line 3.
```

Unclassified Traps

Everything else.

* `require`/`do` trap using returned value

If the file doit.pl has:

```
sub foo {
    $rc = do "./do.pl";
    return 8;
}
print &foo, "\n";
```

And the do.pl file has the following single line:

```
return 3;
```

Running doit.pl gives the following:

```
# perl 4 prints: 3 (aborts the subroutine early)
# perl 5 prints: 8
```

Same behavior if you replace `do` with `require`.

* `split` on empty string with LIMIT specified

```
$string = '';
@list = split(/foo/, $string, 2)
```

Perl4 returns a one element list containing the empty string but Perl5 returns an empty list.

As always, if any of these are ever officially declared as bugs, they'll be fixed and removed.