

NAME

perltrap - Perl traps for the unwary

DESCRIPTION

The biggest trap of all is forgetting to use `warnings` or use the `-w` switch; see *warnings* 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 `ifs` and `whiles`.
- 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 -1` to find their names on your system.

JavaScript Traps

Judicious JavaScript programmers should take note of the following:

- In Perl, binary `+` is always addition. `$string1 + $string2` converts both strings to numbers and then adds them. To concatenate two strings, use the `.` operator.
- The `+` unary operator doesn't do anything in Perl. It exists to avoid syntactic ambiguities.
- Unlike `for...in`, Perl's `for` (also spelled `foreach`) does not allow the left-hand side to be

an arbitrary expression. It must be a variable:

```
for my $variable (keys %hash) {  
    ...  
}
```

Furthermore, don't forget the `keys` in there, as `foreach my $kv (%hash) {}` iterates over the keys and values, and is generally not useful (`$kv` would be a key, then a value, and so on).

- To iterate over the indices of an array, use `foreach my $i (0 .. $#array) {}`. `foreach my $v (@array) {}` iterates over the values.
- Perl requires braces following `if`, `while`, `foreach`, etc.
- In Perl, `else if` is spelled `elsif`.
- `? :` has higher precedence than assignment. In JavaScript, one can write:

```
condition ? do_something() : variable = 3
```

and the variable is only assigned if the condition is false. In Perl, you need parentheses:

```
$condition ? do_something() : ($variable = 3);
```

Or just use `if`.

- Perl requires semicolons to separate statements.
- Variables declared with `my` only affect code *after* the declaration. You cannot write `$x = 1; my $x;` and expect the first assignment to affect the same variable. It will instead assign to an `$x` declared previously in an outer scope, or to a global variable.
Note also that the variable is not visible until the following *statement*. This means that in `my $x = 1 + $x` the second `$x` refers to one declared previously.
- `my` variables are scoped to the current block, not to the current function. If you write `{my $x;} $x;`, the second `$x` does not refer to the one declared inside the block.
- An object's members cannot be made accessible as variables. The closest Perl equivalent to `with(object) { method() }` is `for`, which can alias `$_` to the object:

```
for ($object) {  
    $_->method;  
}
```

- The object or class on which a method is called is passed as one of the method's arguments, not as a separate `this` value.

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 `cs`.
- Shells (especially `cs`) 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 *perls*.
- 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 *perform* 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.

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