NAME

perlstyle - Perl style guide

DESCRIPTION

Each programmer will, of course, have his or her own preferences in regards to formatting, but there are some general guidelines that will make your programs easier to read, understand, and maintain.

The most important thing is to run your programs under the **-w** flag at all times. You may turn it off explicitly for particular portions of code via the `no warnings` pragma or the `$^W` variable if you must. You should also always run under `use strict` or know the reason why not. The `use sigtrap` and even `use diagnostics` pragmas may also prove useful.

Regarding aesthetics of code lay out, about the only thing Larry cares strongly about is that the closing curly bracket of a multi-line BLOCK should line up with the keyword that started the construct. Beyond that, he has other preferences that aren't so strong:

- 4-column indent.

- Opening curly on same line as keyword, if possible, otherwise line up.

- Space before the opening curly of a multi-line BLOCK.

- One-line BLOCK may be put on one line, including curlies.

- No space before the semicolon.

- Semicolon omitted in "short" one-line BLOCK.

- Space around most operators.

- Space around a "complex" subscript (inside brackets).

- Blank lines between chunks that do different things.

- Uncuddled elses.

- No space between function name and its opening parenthesis.

- Space after each comma.

- Long lines broken after an operator (except `and` and `or`).

- Space after last parenthesis matching on current line.

- Line up corresponding items vertically.

- Omit redundant punctuation as long as clarity doesn't suffer.

Larry has his reasons for each of these things, but he doesn't claim that everyone else's mind works the same as his does.

Here are some other more substantive style issues to think about:

- Just because you *CAN* do something a particular way doesn't mean that you *SHOULD* do it that way. Perl is designed to give you several ways to do anything, so consider picking the most readable one. For instance

  ```
  open(FOO,$foo) || die "Can't open $foo: $!";
  ```

  is better than

  ```
  die "Can't open $foo: $!" unless open(FOO,$foo);
  ```

  because the second way hides the main point of the statement in a modifier. On the other

hand

```
    print "Starting analysis\n" if $verbose;
```

is better than

```
    $verbose && print "Starting analysis\n";
```

because the main point isn't whether the user typed **-v** or not.

Similarly, just because an operator lets you assume default arguments doesn't mean that you have to make use of the defaults. The defaults are there for lazy systems programmers writing one-shot programs. If you want your program to be readable, consider supplying the argument.

Along the same lines, just because you *CAN* omit parentheses in many places doesn't mean that you ought to:

```
    return print reverse sort num values %array;
    return print(reverse(sort num (values(%array)))));
```

When in doubt, parenthesize. At the very least it will let some poor schmuck bounce on the % key in **vi**.

Even if you aren't in doubt, consider the mental welfare of the person who has to maintain the code after you, and who will probably put parentheses in the wrong place.

- Don't go through silly contortions to exit a loop at the top or the bottom, when Perl provides the `last` operator so you can exit in the middle. Just "outdent" it a little to make it more visible:

```
    LINE:
 for (;;) {
     statements;
   last LINE if $foo;
     next LINE if /^#/;
     statements;
 }
```

- Don't be afraid to use loop labels--they're there to enhance readability as well as to allow multilevel loop breaks. See the previous example.

- Avoid using `grep()` (or `map()`) or `backticks` in a void context, that is, when you just throw away their return values. Those functions all have return values, so use them. Otherwise use a `foreach()` loop or the `system()` function instead.

- For portability, when using features that may not be implemented on every machine, test the construct in an eval to see if it fails. If you know what version or patchlevel a particular feature was implemented, you can test `$]` (`$PERL_VERSION` in `English`) to see if it will be there. The `Config` module will also let you interrogate values determined by the **Configure** program when Perl was installed.

- Choose mnemonic identifiers. If you can't remember what mnemonic means, you've got a problem.

- While short identifiers like `$gotit` are probably ok, use underscores to separate words in longer identifiers. It is generally easier to read `$var_names_like_this` than `$VarNamesLikeThis`, especially for non-native speakers of English. It's also a simple rule that works consistently with `VAR_NAMES_LIKE_THIS`.

  Package names are sometimes an exception to this rule. Perl informally reserves lowercase module names for "pragma" modules like `integer` and `strict`. Other modules should begin with a capital letter and use mixed case, but probably without underscores due to limitations in primitive file systems' representations of module names as files that must fit into a few sparse bytes.

- You may find it helpful to use letter case to indicate the scope or nature of a variable. For example:

```
$ALL_CAPS_HERE   constants only (beware clashes with perl vars!)
$Some_Caps_Here  package-wide global/static
$no_caps_here    function scope my() or local() variables
```

  Function and method names seem to work best as all lowercase. E.g., `$obj->as_string()`.

  You can use a leading underscore to indicate that a variable or function should not be used outside the package that defined it.

- If you have a really hairy regular expression, use the `/x` modifier and put in some whitespace to make it look a little less like line noise. Don't use slash as a delimiter when your regexp has slashes or backslashes.

- Use the new `and` and `or` operators to avoid having to parenthesize list operators so much, and to reduce the incidence of punctuation operators like `&&` and `||`. Call your subroutines as if they were functions or list operators to avoid excessive ampersands and parentheses.

- Use here documents instead of repeated `print()` statements.

- Line up corresponding things vertically, especially if it'd be too long to fit on one line anyway.

```
$IDX = $ST_MTIME;
$IDX = $ST_ATIME      if $opt_u;
$IDX = $ST_CTIME      if $opt_c;
$IDX = $ST_SIZE       if $opt_s;

mkdir $tmpdir, 0700 or die "can't mkdir $tmpdir: $!";
chdir($tmpdir)      or die "can't chdir $tmpdir: $!";
mkdir 'tmp',  0777 or die "can't mkdir $tmpdir/tmp: $!";
```

- Always check the return codes of system calls. Good error messages should go to STDERR, include which program caused the problem, what the failed system call and arguments were, and (VERY IMPORTANT) should contain the standard system error message for what went wrong. Here's a simple but sufficient example:

```
opendir(D, $dir)  or die "can't opendir $dir: $!";
```

- Line up your transliterations when it makes sense:

```
tr [abc]
   [xyz];
```

- Think about reusability. Why waste brainpower on a one-shot when you might want to do something like it again? Consider generalizing your code. Consider writing a module or object class. Consider making your code run cleanly with `use strict` and `use warnings` (or **-w**) in effect. Consider giving away your code. Consider changing your whole world view. Consider... oh, never mind.

- Try to document your code and use Pod formatting in a consistent way. Here are commonly expected conventions:

  - use `C<>` for function, variable and module names (and more generally anything that can be considered part of code, like filehandles or specific values). Note that function names are considered more readable with parentheses after their name, that is `function()`.

  - use `B<>` for commands names like **cat** or **grep**.

- use `F<>` or `C<>` for file names. `F<>` should be the only Pod code for file names, but as most Pod formatters render it as italic, Unix and Windows paths with their slashes and backslashes may be less readable, and better rendered with `C<>`.

- Be consistent.

- Be nice.