Writing better Perl

Tips and tricks to make Perl easier, faster, and more fun

Kirrily Robert
Founder
Netizen Pty Ltd

1. Introduction

This tutorial is aimed at Perl programmers who have a general understanding of the language, but wish to improve their code in the areas of readability, maintainability, solidity, and so on.

It is assumed that the students understand the following general topics:

- Perl data types (scalars, arrays, and hashes)
- Conditional and looping constructs
- Perl functions and operators
- Basic regular expressions

Students may be currently working as system administrators using Perl for ad-hoc scripting and similar tasks, programmers in the early stages of migrating from other languages, or web developers looking to enhance their understanding of the world's most popular CGI scripting language.

2. Basic programming skills

There are certain basic programming skills which can be applied to Perl to make it more readable and maintainable.

2.1. Indentation and other whitespace

Blocks of code (nested inside braces) should be indented to make it easier to follow the flow of the program:

```
if ($a) {
        foo();
}
```

Whitespace is useful both inside a statement and between statements. Hints on whitespace can be found in **perldoc perlstyle**:

- 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.
- Space around most operators.
- Space around a "complex" subscript (inside brackets).
- Blank lines between chunks that do different things.
- No space between function name and its opening parenthesis.
- Space after each comma.
- Long lines broken after an operator (except "and" and "or").

2.2. Naming conventions

The following simple rules are taken from Netizen's style guide and closely follow Perl "best practice":

- Separate words in names with underscores, except in module names, eg "employee_number"
- UPPER_CASE for globals and constants

- MixedCase for packages and modules
- lower_case for normal variables and subroutines
- The length of the name should correlate to its scope, i.e. temporary variables may be called `$i` but global variables should be called `$MAX_RECORD_COUNT` or similar.

2.3. Scoping

Scoping variables keeps them in their proper place, prevents certain types of bugs, and makes a maintainer's life easier. Don't use global variables unless absolutely necessary! See the section on `my()` below for more information on scoping variables in Perl.

3. Idiomatic Perl

Perl is an unusual language in that it attempts to make things easier for the programmer by making the language messier. In this way it's quite like English. For any given task, "there's more than one way to do it".

Perl has many unusual constructs which, while they may not be known to a novice, are useful for making your code more readable or for abstracting certain algorithms which may otherwise take many lines of minutely detailed code.

This section outlines some of Perl's main weirdnesses and shortcuts which, like spoken language idioms, can help make your code more expressive or succinct.

3.1. Understanding context

The first Perl idiom which needs to be covered is the concept of context. It is essentially a simple concept, but one which often causes difficulties for those who are used to "clean" languages.

Perl has two main contexts in which statements are interpreted: scalar context and list context. If you ask Perl for a scalar, it will give you a scalar; if you ask it for a list, it will give you a list:

```
my @array = ("a", "b", "c");
my $foo = $array[0];                    # gives "a"
my $foo = @array[0];                    # gives ("a")
```

This behaviour if often the cause of mysterious bugs for novice (and even sometimes experienced) Perl programmers. Some Perl functions also return different things in list and scalar context:

```
my $time = localtime();         # gives Sun Jul  2 23:55:05 EST 2000
my @time = localtime();         # gives (second, minute, hour, etc)
print localtime();              # what does this print?
```

Scalar context also has lesser contexts, known as numeric and string contexts. Perl will usually convert most variables to the most useful format:

```
my $foo = "a";
my $bar = 23;

print $foo, $bar, "\n";                 # converts $bar to string

if ($foo == $bar) {                     # numeric context!
        print "Numerically equal.\n";
}

if ($foo eq $bar) {                     # string context!
        print "Stringily equal.\n";
}
```

The `-w` warning flag will warn you if you try to compare a string in numeric context.

3.2. defined, exists and Perl's idea of truth

Perl's idea of truth is different to many other languages. In Perl, the following are false:

- the number zero (`0`)
- the empty string (`""`)
- an undefined value (`undef`)

It is important to understand the process Perl goes through to test for truth, because sometimes it can catch you out.

- First, if the value is not already a string, it is stringified. Strings remain as they are.
- If the string is `""` or `"0"` it is false.

This leads to the strange situation where the string `"0.0"` is TRUE.

But this is an unusual case, seldom seen in the wild. More commonly, confusion arises from the differences between definedness, existence, and truth.

```
my $a = 0;
if ($a) { }                     # false
if (defined $a) { }             # true

my %hash = (
        a       =>      0,
        b       =>      1,
        c       =>      2
);

if ($hash{a}) { }               # false because it is zero
if (exists $hash{d}) { }        # false; no warning
if ($hash{d}) { }               # false; produces warning under -w
```

3.3. foreach

The `foreach` loop may be used to iterate easily through any list:

```
# here’s what C programmers do...
for ($i = 0; $i <= $#array; $i++) {
        print $array[$i];
}

# and here it is in Perl...
foreach (@array) {
        print;
}
```

You can assign each list item to something other than `$_` if you wish:

```
foreach $item (@array) {
        print $item;
}
```

It also works for non-array lists:

```
foreach (1, 2, 3, 4, 5) {
        print $number * $_;
}

foreach $key (keys %hash) {
        print "$key is $hash{$key}\n";
}
```

3.4. The ".." list creation operator

The `..` operator can be used to create a numeric sequence on the fly, which can then be used anywhere a list is used:

```
print 0..9;     # prints 0123456789

# print the first 10 elements in an array
foreach (0..9) {
        print $array[$_];
}
```

3.5. unless and until

Perl implements the "backwards" conditionals, `unless` and `until`. These can make certain conditions more readable:

```
# the C way:
if (! defined($a)) { }

# the Perl way:
unless ($a) { }

# a while loop...
while ($menu_choice ne "q") { }
```

```
# using until...
until ($menu_choice eq "q") { }
```

3.6. split and join

The `split` and `join` functions are extremely useful for turning scalars into lists and vice versa:

```
# convert an /etc/passwd entry to an array
my $user = qq(skud:x:1000:1000:Kirrily Robert„,:/home/skud:/usr/bin/tcsh);
my @fields = split /:/, $user;

# manipulate the array
$fields[$#fields] = "/bin/bash";

# convert it back...
my $new_user = join ":", @fields;
```

3.7. map and grep

The `map` function can be used to perform an action on every element of an array:

```
# the long way
my @hex_array;
my @doubled_array;
foreach (@array) {
        push @hex_array, hex($_);
        push @doubled_array, $_ * 2;
}

# using map
my @hex_array = map(hex, @array);
my @doubled_array = map {$_ * 2} @array;
```

The `grep` function likewise provides a quick way of searching for elements in an array.

```
# the long way
my @matches;
foreach (@array) {
        if (/$string/) {
                print;
                push @matches;
        }
}

# using grep
print grep(/$string/, @array);
my @matches = grep /$string/, @array;

# grep can also use a block rather than a regexp:
my @longwords = grep { length{$_} > 8 } @array;
```

3.8. Regular expressions

Regular expressions are one of Perl's most powerful features, and it is worthwhile learning how to use them effectively. The first step to this, which is not covered in this tutorial, is having a good understanding of how the various metacharacters and other regexp tricks work (if you want to know more about that, **perldoc perlre** or read "Mastering Regular Expressions" (O'Reilly's Owl book)).

The second, and sometimes more difficult step is making your regular expressions readable.

3.8.1. Alternative delimiters

Alternative delimiters may be used for matching and/or substitution:

```
# bad
if (/http:\/\/netizen.com.au\/~skud\//) { }

# better
if (m!http://netizen.com.au/~skud/!) { }
if (m(http://netizen.com.au/~skud/)) { }

# substitution the ugly way
s/\/usr\/local\/bin/\/usr\/bin/g;

# less ugly
s!/usr/local/bin!/usr/bin!g
s{/usr/local/bin}{/usr/bin}g
```

3.8.2. Commented regexps

You can add comments and whitespace into your regexps using the `/x` (extended regexp) modifier.

```
# check for validly formed download URL
if ($url_string =~ m!
        (http|ftp)://                   # protocol part
        [^/]+                           # host part
        .*                              # everything else
!x) { }
```

As far as I am aware, these two techniques are unique to Perl.

4. Solidity

A solid program is one which will not break easily. Here are a few tips for making your Perl programs more solid:

4.1. Version dependence

If your program depends on a certain version of Perl to run, you can check for it using the `require` function.

```
require 5.005;
```

4.2. Warnings and diagnostics

Warnings can be turned on using the `-w` switch to the Perl interpreter, either on the command line or in the `#!` line at the top of the script.

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

print $foo;             # gives a warning as $foo is undefined

my %hash = (
        a => 1,
        b => 2,
        c => 3
);
print $hash{d};         # gives a warning as $hash{d} doesn’t exist
```

4.3. The strict pragma and scoping

The `strict` pragma can be used to enforce pre-declaration and scoping of variables (which is achieved via the `my()` function) and to catch certain other possible problems in your code:

```
use strict;

my $a = 1;
print $a;               # prints 1

if ($a) {
        my $b = 2;
        print $b;       # prints 2
        my $a = 3;      # creates new $a in this scope
        print $a;       # prints 3
}

print $a;               # prints 1
print $b;               # fails; $b is not in scope
```

A program which "uses strict" is much less likely to suffer from bugs resulting from mis-typed or poorly remembered variable names.

The strict pragma will also check references and subroutines. See **perldoc strict** and/or the "Pragmatic Modules" section in **perldoc perlmodlib** for a full explanation of what it does.

4.4. Taint checking

Taint checking forces you to check user input for malicious data. It can be turned on using the `-T` switch.

```
#!/usr/bin/perl -wT
```

```
my $filename = $ARGV[0];              # first command line argument

unlink $filename;                     # fails; $filename is tainted

# data can be untainted by referring to a substring from a match

if ($filename =~ /^([^/]+)$/) {
        unlink $1;
} else {
        print "You may not use slashes in filenames.\n";
}
```

4.5. Using built-in functions rather than system()

Perl has many built-in functions which can be used instead of calling `system()`. Some examples:

```
system("chmod 755 myfile");
chmod 0755, "myfile";

system("rm myfile");
unlink "myfile";
```

The benefit of using the Perl functions is that they are less platform dependent and it is easier to trap errors.

4.6. Testing system calls

Whenever you make a system call, whether it is through `system()` or through a native Perl function, you should always check for success or failure.

```
open INPUT, $input_file or die "Can’t open input file: $!";
unlink "myfile" or warn "Can’t unlink file: $!";
```

5. Reinventing the wheel (and how to avoid it)

There are a number of modules which come standard with Perl, or which can be obtained from CPAN, which will help you avoid most of the drudge work of performing common tasks. The following short list gives just a few small examples that I frequently find useful; you will no doubt find many others if you explore CPAN's module list.

5.1. Getopt::Std and Getopt::Long

Used to accept and parse command line options

```
use Getopt::Std;
getopt("hvq", \%args);  # -h, -v, -s accepted into a hash
getopt("hvq");          # -h, -v, -q, sets $opt_h, $opt_v, $opt_q
                        # note: doesn't work well with strict pragma
getopts("s:");          # makes it accept -s ARG
print $VERSION if $opt_v;

use Getopt::Long;

GetOptions(     "help"          =>      \$help,
                "version"       =>      \$version,
                "quiet"         =>      \$quiet );

print $VERSION if $version;
```

5.2. File::Find

This can be used to recurse down directories easily:

```
use File::Find;

find(\&wanted, $directory);

sub wanted {
        print if /\.(old|bak)$/;                # find backup files
}
```

This replaces the much more painful technique of recursing by hand using `readdir()` and other such functions.

5.3. Text::Template

Fill in a template such as a configuration file, email, or HTML page.

```
use Text::Template;

my $template = new Text::Template(TYPE => ’FILE’, SOURCE => $myfile);
print $template->fill_in(HASH => $hashref);
```

This is much easier than writing your own substitution engine to replace keywords. The Text::Template documentation says:

When people make a template module like this one, they almost always start by inventing a special syntax for substitutions. For example, they build it so that a string like %%VAR%% is replaced with the value of $VAR. Then they realize the need extra formatting, so they put in some special syntax for formatting. Then they need a loop, so they invent a loop syntax. Pretty soon they have a new little template language. This approach has two problems: First, their little language is crippled. If you need to do something the author hasn’t thought of, you lose. Second: Who wants to learn another language? You already know Perl, so why not use it?

5.4. Mail::Mailer

Easily send Internet email.

```
use Mail::Mailer;

my $mail = new Mail::Mailer;

$mail->open(
        To      =>      $recipient,
        Subject =>      $subject,
        From    =>      $sender
);

print $mail <<"END";
```

```
This is your email body.
END

$mail->close();
```

This is much less error prone than the usual technique of opening a pipe to sendmail, etc. One major benefit is that it will try different techniques for sending mail, including clients such as **mailx**, mail servers such as **sendmail**, or as a last result, raw SMTP.

5.5. LWP::UserAgent

Get web pages via HTTP. The simplest way is actually to use LWP::Simple, like so:

```
use LWP::Simple;
unless (defined (my $content = get $URL)) {
        die "Could not get $URL\n";
}
```

No more opening sockets and connecting to port 80!

6. Usability and maintainability

6.1. Using English

The English module, which comes with Perl, can give you COBOL-style verbose names for Perl's special variables. For instance, the following statements are equivalent:

```
use English;

$/ = "\n\n";
$INPUT_RECORD_SEPARATOR = "\n\n";
```

6.2. Error messages

`warn()` and `die()` can be used to print warnings or fatal errors to STDERR. Here is one common usage:

```
open (INPUT, $inputfile) or die("Can’t open input file: $!");
```

The Carp module, which comes standard with Perl, can produce even more useful error messages. This is from its documentation:

The Carp routines are useful in your own modules because they act like die() or warn(), but report where the error was in the code they were called from. Thus if you have a routine Foo() that has a carp() in it, then the carp() will report the error as occurring where Foo() was called, not where carp() was called.

CGI::Carp is another module which may be of particular use to those who write a lot of CGI; it can output error messages to the httpd error log, to a special logfile, or to the browser. The latter option, in particular, makes debugging CGI much faster.

6.3. Documentation

Needless to say, documentation is a very necessary and much ignored part of most Perl programs.

The first and simplest type of documentation is the simple help message. A script which outputs a short synopsis of its parameters when it is given the `-h`, `--help`, or any invalid option, is much more usable than one which requires the user to RTFS.

Perl also provides a system for writing and reading documentation embedded in scripts. This is called "POD", or "plain old documentation", and is most commonly used in Perl modules. Instructions for writing POD can be found in the **perlpod** man page. Meanwhile, here’s a short example:

```
=pod

=head1 NAME

My::Module - do some stuff

=head1 SYNOPSIS
```

```
use My::Module

=head1 DESCRIPTION

This is the descrip-
tion of My::Module.  This is how you make text B<bold>,
I<italic>, or C<literal code>.

=cut
```

7. Summary

In summary: you owe it to yourself, and your successors, to write Perl that is at least slightly readable and maintainable. Perl programming is meant to be enjoyable, and it's easy to make it more so by doing it well.

The techniques outlined above, while by no means comprehensive, should give you a good start towards writing better Perl.

8. Further reading

These are just a few personal recommendations:

- The Perl Cookbook, by Tom Christiansen and Nathan Torkington. Published by O'Reilly.
- Effective Perl Programming, by Joseph N. Hall. Published by Addison-Wesley
- Perl's excellent man pages, all of them. Really.
- ... including the Perl FAQ, which answers many questions about how to do things "the Perl way" and how to avoid reinventing the wheel, all in good idiomatic Perl
- In defense of coding standards, by Kirrily Robert, published by Perl.com (http://www.perl.com/pub/2000/01/CodingStandards.html) in January 2000.