9

Running and Debugging Perl

By now, we've essentially covered the whole of the core Perl language. We've yet to see how we can use pre-packaged modules to accomplish a great many common tasks, including applying Perl to networking, CGI and database manipulation. But right now, we've finished as much of the language as you'll need to know for pretty much everything you'll want to do with Perl. Congratulations for getting this far!

You should also be getting used to analyzing the problem you want to solve, breaking it down into component parts, and thinking about how to explain those parts to the computer in a language it can understand. That's not all, however.

Everyone makes mistakes. It's a simple fact of life, and programming is just the same. When you write programs, you will make mistakes. As we mentioned in the first chapter, the name for a mistake in programming is a **bug**, and the process of removing bugs is called **debugging**. After breaking down your ideas and writing the code, you'll come to the next two phases of software development: testing and debugging.

In this chapter, we'll see how Perl helps us with these stages. In particular, we'll cover the following areas:

- **Error Messages**
  How the perl interpreter tells you you've used the language incorrectly.
- **Diagnostic Modules**
  What modules can help us isolate and understand problems with our code.
- **Perl Command Line Switches**
  Creating test programs using the perl command line.
- **Debugging Techniques and the Perl Debugger**
  How to remove the problems that we've found.

By the end of this chapter, you should be able to recognize, diagnose, and hopefully fix any programming errors you make. We'll also look at how to construct test cases and quick one-line programs on the perl command line.

Error Messages

There are two types of mistake you can make when programming: a syntax error and a logic error. A **syntax error** is something like a typo or a result of misunderstanding how to use the language, meaning that your code doesn't actually make sense any more. Since your code isn't properly written in Perl, perl can't understand it and complains about it.

A **logic error**, on the other hand, is where the instructions you give make perfect sense, but don't actually do what you think they ought to. This type of error is far more dastardly to track down but there are ways and means to do so. For the time being, though, we'll start by looking at the way Perl detects and reports syntax errors:

Try It Out : Examining Syntax Errors

Let's create a few syntax errors, and see how Perl reports them to us. Take the following program, for example:

```
#!/usr/bin/perl
# errors.plx
use warnings;
use strict;

my $a;
print "Hello, world."
$a=1;
if ($a == 1 {
   print "\n";
}
```

As you should be able to see if you look carefully, this contains a number of mistakes. This is what Perl makes of it:

```
>perl errors.plx
Scalar found where operator expected at errors.plx line 8, near "$a"
      (Missing semicolon on previous line?)
syntax error at errors.plx line 8, near "$a"
syntax error at errors.plx line 9, near "1 {"
Execution of errors.plx aborted due to compilation errors.
>
```

How it Works

What's Perl complaining about? Firstly, it sees something up on line 8:

```
$a=1;
```

Well, there's nothing wrong with that. That's perfectly valid code. When we're trying to track down and understand syntax errors, the key thing to remember is that the line number Perl gave us is *as far as it got* before realizing there was a problem – that doesn't necessarily mean that the line itself has a problem. If, for instance, we miss out a closing bracket, Perl may go all the way to the end of the file before complaining. In this case, though, Perl gives us an additional clue:

(Missing semicolon on previous line?)

In fact, this is exactly the problem:

```
print "Hello, world."
```

Line 7 doesn't end with a semicolon. But what of the error message, 'Scalar found where operator expected'? What does this mean? Like all of Perl's error messages, it means exactly what it says. Perl found a scalar where it thought there should be an operator. But why? Well, Perl had just finished processing a string, which was fed to `print`. But since there wasn't a semicolon, it was trying to find a way to continue the statement. The only way to continue would be to have an operator to link the string with something else: the concatenation operator, for instance, to connect it to another scalar. However, instead of such an operator, Perl found the scalar `$a`. Since you can't put a string right next to a variable, Perl complains, and as there's no way for this to make sense, it also gives us a 'syntax error'.

The next problem is in line 9:

```
if ($a == 1 {
```

Here we have no clue to help us track down the bug. It's a syntax error pure and simple, and we can fix it easily by providing the missing bracket. It should, of course, look like this:

```
if ($a == 1) {
```

Syntax Error Checklist

Tracking down syntax errors can be troublesome, but it's a skill that comes with practice. Most of the errors you're likely to experience are going to fall into one of the six categories below:

Missing Semicolons

We've seen this already, and it's probably the most common syntax error there is. Every statement in Perl, unless it's at the end of a block, should finish with a semicolon. Sometimes you'll get the helpful hint we got above:

(Missing semicolon on previous line?)

but otherwise you've just got to find it yourself. Remember that the line number you get in any error message may well not be the line number the problem occurs on – just when the problem is detected.

Missing Open/Close Brackets

The next most common error comes when you forget to open or close a bracket or brace. Missed closing braces are the most troublesome, because Perl sometimes goes right the way to the end of the file before reporting the problem. For example:

```
#!/usr/bin/perl
# braces.plx
use warnings;
use strict;
```

```
if (1) {
   print "Hello";

my $file = shift;
if (-e $file) {
   print "File exists.\n";
}
```

This will give us:

```
>perl braces.plx
Missing right curly or square bracket at braces.plx line 12, at end of line
syntax error at braces.plx line 12, at EOF
Execution of braces.plx aborted due to compilation errors.
>
```

The problem is, our missing brace is only at line 7, but Perl can't tell that. To find where the problem is in a large file, there are a variety of things you can do:

- Indent your code as we have done to make the block structure as clear as possible. This won't affect what perl sees, but it helps *you* to see how the program hangs together, making it more readily obvious when this sort of thing happens.
- Deliberately leave out semicolons where you think a block should end, and you'll cause a syntax error more quickly. However, you'll need to remember to add the semicolon if you add extra statements to the block.
- Use an editor which helps you out: Editors like `vi` and `emacs` automatically flash up matching braces and brackets (called **balancing**) and are freely available for both UNIX and Windows.

We'll also be looking at some more general techniques for tracking down bugs later on in this chapter.

Runaway String

In a similar vein, don't forget to terminate strings and regular expressions. A runaway string will cause a cascade of errors as code looks like strings and strings look like code all the way through your program. If you're lucky though, Perl will catch it quickly and tell you where it starts – miss off the closing `"` in line 7 of the above example, and Perl will produce this message amongst the rest of the mess:

```
(Might be a runaway multi-line "" string starting on line 7)
```

This is also particularly pertinent when you're dealing with here-documents. Let's look again at the example we saw in Chapter 2:

```
#!/usr/bin/perl
#heredoc.plx
use warnings;
print<<EOF;

This is a here-document. It starts on the line after the two arrows,
and it ends when the text following the arrows is found at the beginning
of a line, like this:

EOF
```

Since perl treats everything between `print<<EOF;` and the terminator `EOF` as plain text, it only takes a broken terminator for perl to interpret the rest of your program as nothing more than a long string of characters.

Missing Comma

If you forget a comma where there should be one, you'll almost always get the 'Scalar found where operator expected' message. This is because Perl is trying to connect two parts of a statement together and can't work out how to do it.

Brackets Around Conditions

You need brackets around the conditions of `if`, `for`, `while`, and their English negatives `unless`, `until`. However, you don't need brackets around the conditions when using them as statement modifiers.

Barewords

If an error message contains the word 'bareword', it means that Perl couldn't work out what a word was supposed to be. Was it a scalar variable and you forgot the type symbol? Was it a filehandle used in a funny context? Was it an operator or subroutine name you spelled wrong? For example, if we run:

```
#!/usr/bin/perl
#bareword.plx
use warnings;
use strict;

Hello;
```

perl will tell us:

```
>perl bareword.plx
Bareword "Hello" not allowed while "strict subs" in use at bareword.plx line 5.
Execution of braces.plx aborted due to compilation errors.
>
```

We'll see more in the section on barewords in `use strict` below.

Diagnostic Modules

Hopefully, I've already drummed into you the importance of writing `use strict` and `use warnings` in your code. Now it's time to explain what those, and other modules like them, actually do.

As we'll see in the next chapter, `use` introduces an external module, while `warnings` and `strict` are both standard Perl modules that come with the Perl distribution. They're just ordinary Perl code. The special thing about them is that they fiddle with internal Perl variables, which will alter the behavior of the perl interpreter.

Strictly speaking these are **pragmas** (or, for the linguistically inclined, *pragmata*) rather than modules. These have all lower-case names and are particularly concerned with altering the operation of perl itself, rather than providing you with ready-made code.

use warnings

The `warnings` pragma changes the way perl produces warnings. Ordinarily, there are a number of warnings that you can turn on and off, categorized into a series of areas: syntactic warnings, obsolete ways of programming, problems with regular expressions, input and output, and so on.

Redeclaring Variables

By default, all warnings are turned off. If you merely say `use warnings`, everything is turned on. So, for example, without specifying `use warnings`, the following code will execute without issue:

```
#!/usr/bin/perl
# warntest.plx
# add 'use warnings;' command here

my $a = 0;
my $a = 4;
```

However, with `use warnings` specified after the filename comment, here is what perl tells you:

```
>perl warntest.plx
"my" variable $a masks earlier declaration in same scope at warntest.plx line 6.
>
```

What does this mean? It means that in line 6, we declared a new variable `$a`. If you remember, `my` creates a completely new variable. However, we already have a variable `$a`, which we declared in line 5. By re-declaring it in line 6, we lose the old value of `0`. This is a warning in the 'misc' category.

Misspelling Variable Names

Let's see another common cause of error - misspelling variable names:

```
#!/usr/bin/perl
# warntest2.plx
# add 'use warnings;' command here

my $total = 30;
print "Total is now $total\n";
$total += 10;
print "Total is now $tutal\n";
```

Without warnings, we see this:

```
> perl warntest2.plx
Total is now 30
Total is now
```

Why has our variable lost its value? Let's turn on warnings and run this again. Now we get:

```
> perl warntest2.plx
Name "main::tutal" used only once: possible typo at warntest2.plx line 8.
Total is now 30
Use of uninitialized value in concatenation (.) at warntest2.plx line 8.
Total is now
```

Aha! A warning in the 'once' category has been fired, telling us that we've only used the variable `tutal` once. Obviously, we've misspelled `total` here.

That's enough to help us track down and fix the problem, but what about the other error: `$tutal` certainly had an uninitialized value, but where is the concatenation? We didn't use the `.` operator – however, perl did. Internally, perl understands `"something $a"` to be `"something ".$a`. Since the `$a` in this case was undefined, perl complained.

The Scope of use warnings

The `warnings` pragma is **lexically scoped**, so its effects will last throughout the same block of code as a `my` variable would – that is, within the nearest enclosing braces or the current file. For instance, the following program has warnings throughout:

```
#!/usr/bin/perl
# warntest3.plx
use warnings;

{
   my @a = qw(one , two , three , four);
}
my @b = qw(one , two , three , four);
```

Therefore perl responds with the following warnings, in the `qw` category:

```
>perl warntest3.plx
Possible attempt to separate words with commas at warntest3.plx line 6.
Possible attempt to separate words with commas at warntest3.plx line 8.
>
```

reminding us that since `qw()` automatically changes separate words into separate elements, we don't need to separate them with commas.

If you really **do** want commas as some elements of your array, you may turn warnings off by saying `no warnings`. In the following program, warnings are only turned on for the code outside the brackets:

```
#!/usr/bin/perl
# warntest4.plx
use warnings;

{
   no warnings;
   my @a = qw(one , two , three , four);
}
my @b = qw(one , two , three , four);
```

Now perl will only give the one warning, for the second array:

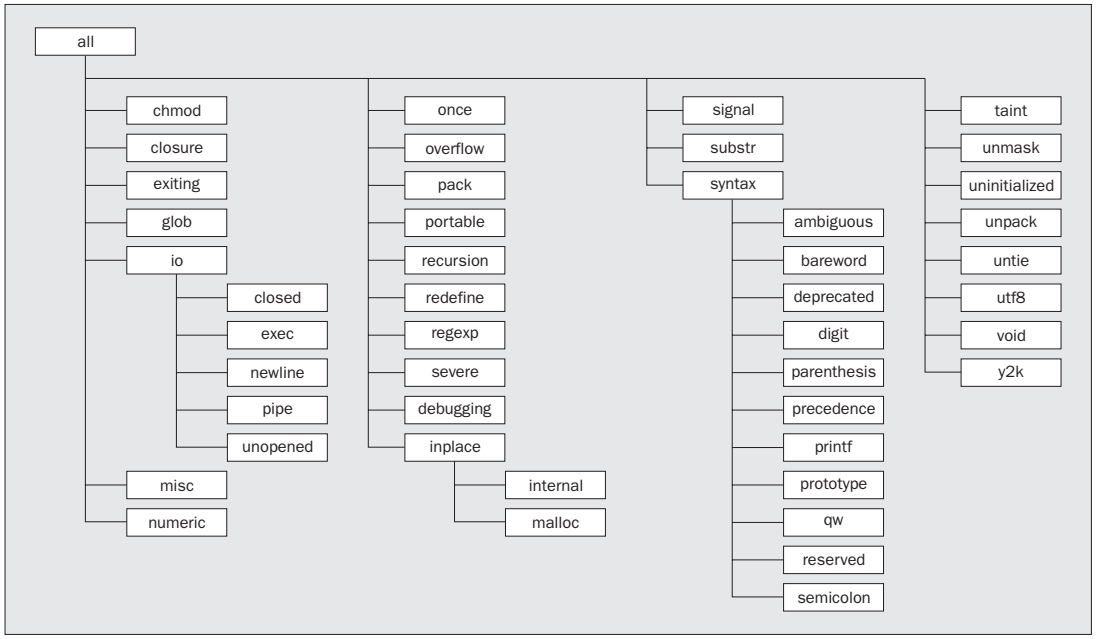
```
> perl warntest4.plx
Possible attempt to separate words with commas at warntest3.plx line 9.
>
```

To turn off or on certain classes of warnings, give them as a list after the `use` or `no warnings`. So, in this case, to just turn off the warning about `qw` but leave the others untouched, you would write:

```
#!/usr/bin/perl
# warntest4.plx
use warnings;

{
   no warnings "qw";
   my @a = qw(one , two , three , four);
}
my @b = qw(one , two , three , four);
```

The categories of warnings you can turn on and off are organized hierarchically as follows, and the warnings they cover are detailed in the `perldiag` documentation:



use strict

You should also know by now that `use strict` forces you to declare your variables before using them. In fact, it controls three areas of your programming: variables, references, and subroutines.

Strict on Variables

First, we'll look at the variables. When `use strict` is applied, a variable must either be declared lexically (using `my $var`) and belong to a block or file, or be declared globally, to be available in every part of the program. You can do this either by using `our $var` (in the same way as `my`) or by specifying its full name, `$main::var`. We'll see where the `main` comes from in the next chapter.

You may see another way of declaring a global, `use vars '$var'`, which does exactly the same as `our`. `our` was introduced in Perl 5.6.0 and is recommended for use, wherever backward compatibility isn't an issue.

If `use strict` applies and you have not used one of these forms of declaration, Perl will not allow you to run the program:

```
#!/usr/bin/perl
# strictvar.plx
use warnings;
use strict;

my $a = 5;
$main::b = "OK";
our $c = 10;
$d = "BAD";
```

The first three variables are fine, but:

```
>perl strictvar.plx
Global symbol "$d" requires explicit package name at strictvar.plx line 9.
Execution of strictvar.plx aborted due to compilation errors.
>
```

To fix this, you just need to use one of the above ways of declaring the variable. Here's an important lesson in debugging: don't turn off the warnings – **fix the bug**. This is especially important when we come to our next cause of problems, references.

Strict on References

One thing novice programmers often want to do is to construct a variable whose name is generated from the contents of another variable. For instance, you're totalling numbers in a file with several sections. Each time you come to a new section, you want to keep the total in another variable. So, you might think you want `$total1`, `$total2`, `$total3`, and so on, with `$section` pointing to the current section. The problem then is to create a variable out of `"total"` plus the current value of `$section`. How do you do it?

- **Honest answer**:
  You can say `${"total".$section}`.
- **Better answer**:
  Don't do it. In such cases, it's almost always better to use a hash or an array. Here, since the sections are numeric, you'd use an array of totals. It's far easier to say `$total[$section]`. More generally, if your sections are named, you'd use a hash, `$total{$section}`.

Why? Well, the most obvious reason is because you know how to use hashes and arrays, and when the question was asked, you didn't know how to construct a variable by name. Use what you know! Don't try and be too clever if there's a simple solution. Constructing these **symbolic references**, as they are known, can play havoc with any of your variables.

Suppose you're making a variable not out of `${"total".$section}` but `${$section}` where `$section` is read from the file. If reading the section name goes horribly wrong, you may have `$section` become one of Perl's special variables, either causing an error or creating weird behavior later in your program – arrays may suddenly stop working, regular expression behavior may become unpredictable, and so on. This kind of thing is a nightmare to debug.

Even if it goes right, there's no guarantee that `$section` won't contain a name you're using somewhere else in the program. A variable name you're using may be blown away at any moment by something outside your program. This isn't a pretty situation to get into, and `use strict` stops you from getting into it by disallowing the use of symbolic references.

Strict on Subroutines

Last, but not least, `use strict` disallows 'poetry optimization', which lets you use barewords as strings. This means if you want to use the name of a subroutine without brackets, you must declare the subroutine first. For example:

```
#!/usr/bin/perl
# strictsubs1.plx
use warnings;
use strict;

$a = twelve;
sub twelve { return 12 }
```

blows up with an error:

```
>perl strictsubs1.plx
Bareword "twelve" not allowed while "strict subs" in use at strictsubs1.plx line 6
Execution of strictsubs1.plx aborted due to compilation errors.
>
```

However, this is okay. You'll get the query 'Name "main::a" used only once: possible typo' but that's simply because we've declared `$a` and then not used it. We'll come back to this error in a minute:

```
#!/usr/bin/perl
# strictsubs2.plx
use warnings;
use strict;

sub twelve { return 12 }
$a = twelve;
```

Of course, you can always get round the limitation on barewords, simply by not using them. A subroutine name with parentheses is always OK:

```
#!/usr/bin/perl
# strictsubs3.plx
use warnings;
use strict;

sub twelve { return 12 }
$a = twelve();
```

These three areas – variables, symbolic references and subroutines – are split into categories just like the warnings. These are `vars`, `refs`, and `subs` respectively.

As before, `use strict` turns on all checks. You can turn on and off all or individual checks on a lexical basis just as you could with `use warnings`:

```
#!/usr/bin/perl
# nostrict.plx
use warnings;
use strict;

our $first = "this";
our $second = "first";
our $third;

{
   no strict ('refs');
    $third = ${$second};
}

print "$third\n";
```

>**perl nostrict.plx**
Name "main::first" used only once: possible typo at nostrict.plx line 6.
this
>

The warnings have been turned off for our symbolic link, but again we get that warning about only explicitly using `$first` once, even though we have indirectly used it again. This is a useful reminder of how warnings work: perl will check to see that the code *looks* structurally sound, but won't actually calculate runtime values or resolve variables. If it did, it would have picked up above on `${$second}` being resolved as `$first`.

> **Don't turn off these checks simply because they stop your program from running. You should always find a way to fix the program so as to satisfy them.**

use diagnostics

There's another pragma that may help you while debugging. `use diagnostics` will show you not only an error message or warning but also the explanatory text from the `perldiag` documentation page. For instance:

```
#!/usr/bin/perl
# diagtest.plx
use warnings;
use strict;
use diagnostics;

my $a, $b = 6;
$a = $b;
```

should give something like:

>perl **diagtest.plx**
Parentheses missing around "my" list at diagtest.plx line 7 (#1)

(W parenthesis) You said something like

my $foo, $bar = @_;

```
    when you meant

        my ($foo, $bar) = @_;

    Remember that "my", "our", and "local" bind tighter than comma.
>
```

This is helpful when you're debugging but remember that when `use diagnostics` is seen, the entirety of the `perldiag` page has to be read into memory, which takes up some time. It's a good idea to use it when writing a program and then remove it when you're done.

Alternatively, there's a standalone program called `splain`, which explains Perl's warnings and error messages in the same way. Simply collect up the output of your program and paste it to `splain`. If you're going to use a pipe, remember that warnings end up on standard error, so you'll have to say `perl myprogram.plx | 2>&1 | splain` to feed standard error there, too. Note that `splain` won't work on Windows.

Perl Command Line Switches

All of our programs so far have started with this line, the 'shebang' line:

```
#!/usr/bin/perl
```

and we've called our program by saying:

>perl program.plx

or possibly on UNIX with:

>./program.plx

The primary purpose of that first line is to tell UNIX what to do with the file. If we say `./program.plx`, this just says 'run the file `program.plx`', it's the 'shebang' line that says how it should be run. It should be passed to the file `/usr/bin/perl`, which is where the Perl interpreter will usually live.

However, that's not all it does, and it isn't just for UNIX: Perl reads this line itself and looks for any additional text, in the form of **switches**, which notify Perl of any special behavior it should turn on when processing the file. If we call the Perl interpreter directly on the command line, by saying `perl program.plx`, we can also specify some switches before Perl even starts looking at the file in question.

Switches all start with a minus sign and an alphanumeric character, and must be placed after `perl` but before the name of the program to be run. For instance, the switch `-w`, which is roughly equivalent to `use warnings;` can be specified in the file, like this:

```
#!/usr/bin/perl -w
# program.plx
...
```

or on the command line like this:

```
>perl -w program.plx
```

This allows us to change Perl's behavior either when writing the program or when running it. Some switches can only be used on the command line. By the time perl has opened and read the file, it may be too late to apply the behavior. This is most clearly illustrated in the case of the `-e` switch, which we'll be taking a look at next.

There are two major types of switch: those that take an argument and those that do not. `-w` does not take an argument, and neither does `-c`. (We'll see what `-c` does very soon.) If you want to specify both switches, you can either put them one after the other, `-w -c`, or combine them in a **cluster**, by saying `-wc`.

For switches that take an argument, such as `-i`, the argument must directly follow the switch. So, while you can combine `-w`, `-c`, and `-i00` as `-wci00`, you may not say `-i00wc`, as the `wc` will be interpreted as part of the argument to `-i`. You must either put switches that take an argument at the end of a cluster or separate them entirely.

-e

The most commonly used switch is `-e`. This may only be used on the command line, because it tells perl not to load and run a program file but to run the text following `-e` as a program. This allows you to write quick Perl programs on the command line. For example, the very first program we wrote can be run from the command line like this:

```
>perl -e 'print "Hello world\n";'
Hello world
>
```

Notice that we surround the entire program in single quotes. This is because, as we saw when looking at `@ARGV`, the shell itself splits up the arguments on the command line into separate words. Without the quotes, our program would just be `print`, with `"Hello world\n"` as the first element of `@ARGV`.

There are two problems with this. First, we can't put single quotes inside our single quotes, and second, some operating systems' shells prefer you to use double rather than single quotes around your program. They then have differing degrees of difficulty coping with quotes in the program.

For instance, DOS, Windows and so on, will want to see this:

```
>perl -e "print \"Hello world\n\";"
```

You can get around most of this by judicious use of the `q//` and `qq//` operators. For instance, you could say `perl -e 'print qq/Hello world\n/;'`, which easily translates to a DOS-acceptable form as `perl -e "print qq/Hello world\n/;"`. Note that on UNIX systems, single quotes are usually preferable, as they prevent the shell interpolating your variables.

In the following examples, we'll be showing examples in single-quoted format. If you're using Windows, just convert them to double-quoted format as described above.

This technique is most commonly used for two purposes:

- To construct quick programs in conjunction with some of the other switches we'll see below
- To test out little code snippets and check how Perl works.

For example, if I wasn't sure whether an underscore would be matched by `\w` in a regular expression, I'd write something like this to check:

```
> perl -e 'print qq/Yes, it's included\n/ if q/_/ =~ /\w/;'
Yes, it's included
>
```

It's often quicker to do this than to go hunting through books and online documentation trying to look it up. As Larry Wall says, 'Perl programming is an empirical science'. You learn by doing it. If you're not sure about some element of Perl, get to a command line and try it out!

-n and -p

As mentioned above, you can combine `-e` with other switches to make useful programs on the command line. The most common switches used in this way are `-n` and `-p`. These are both concerned with reading `<ARGV>`. In fact, `-n` is equivalent to this:

```
while (<>) { "..your code here.." }
```

We can use this to produce programs for scanning through files, searching for matching lines, changing text, and so on. For example, here's a one-liner to print out the subject of any new items of mail I have, along with whom the mail is from:

Try It Out : New Mail Check

All the incoming mail arrives in a file called `Mailbox` on my computer. Each piece of mail contains a header, which contains information about it. For instance, here's part of the header from an email I sent to `perl5-porters`:

```
Date: Mon, 3 Apr 2000 14:22:03 +0900
From: Simon Cozens <simon@cozens.net>
To: perl5-porters@perl.org
Subject: [PATCH] t/lib/b.t
Message-ID: <20000403142203.A1437@SCOZENS>
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
X-Mailer: Mutt 1.0.1i
```

As you can see, each header line consists of some text, then a colon and a space, then some more text. If we extract the lines that start `Subject:` and `From:`, we can summarize the contents of the mailbox.

Here's how to do it on the command line:

```
>perl -ne 'print if /^(Subject|From): /' Mailbox
From: Simon Cozens <simon@brecon.co.uk>
Subject: [PATCH] t/lib/b.t
>
```

How It Works

To extract the relevant lines, we could write a program like this:

```
#!/usr/bin/perl
use warnings;
use strict;

open INPUT, "Mailbox" or die $!;
while (<INPUT>) {
   print if /^(Subject|From): /;
}
```

However, that's a lot of work for a little job, and Perl was invented to make this sort of thing easy. Instead we use the `-n` flag to give us a `while(<>)` loop and `-e` to provide the remaining line. Perl internally translates our one-line incantation to this:

```
LINE: while (defined($_ = <ARGV>)) {
    print $_ if /^(Subject|From): /;
}
```

As you may suspect, we're not confined to just printing text with these one-liners. Indeed, we can use this to modify parts of a file. Let's say we had an old letter file `newyear.txt` containing this text:

```
Thank you for your custom throughout the previous year. We
look forward to facing the challenges that 1999 will bring us,
and hope that we will continue to serve you this year as well.

All our best wishes for a happy and prosperous 1999!
```

We could use perl to print an updated version of it as follows:

```
>perl -ne 's/1999/2000/g; print' newyear.txt
Thank you for your custom throughout the previous year. We
look forward to facing the challenges that 2000 will bring us,
and hope that we will continue to serve you this year as well.

All our best wishes for a happy and prosperous 2000!
>
```

Of course, we're only printing the changed version to `STDOUT`. We could go the next logical step and use redirection to save this output to a file instead, as we saw in Chapter 6.

```
>perl -ne 's/1999/2000/g; print' newyear.txt >changedfile.txt
>
```

Since this is a pretty common operation – 'do something to the incoming data and print it out again' – perl lets use the `-p` flag instead of `-n` to automatically print out the line once we're finished. We can therefore save ourselves a valuable few keystrokes by saying this:

```
>perl -pe 's/1999/2000/g' newyear.txt
```

As you saw from the translation, these are ordinary loops, and we can use `next` and `last` on them as usual. To print out only those lines that don't start with a hash sign (#) we can say this:

```
>perl -ne 'next if /^#/; print' strictvar.plx
use warnings;
use strict;

my $a = 5;
$main::b = "OK";
our $c = 10;
$d = "BAD";
>
```

Note that we don't, and actually **can't** say:

```
>perl -pe 'next if /^#/'
```

This is because `-p` uses a special control structure, `continue`, translating internally to this:

```
LINE: while (defined($_ = <ARGV>)) {
    "Your code here";
} continue { print $_;}
```

Anything in a `continue { }` block will always get executed at the end of an iteration – even if `next` is used (although is still by-passed by `last`).

-c

`-c` stops perl from running your program – instead, all it does is check that the code can be compiled as valid Perl. It's a good way to quickly check that a program has no glaring syntax errors. It also loads up and checks any modules that the program uses, so you can use it to check that the program has everything it needs:

```
>perl –ce 'print "Hello, world\n";'
-e syntax OK
>perl -ce 'print ("Hello, world\n"));'
syntax error at -e line 1, near "))"
-e had compilation errors.
>
```

Be careful though, because this won't necessarily prove that your program will run properly – it checks that your program is grammatically correct, but not whether it makes sense. For instance, this looks fine:

```
>perl –ce 'if (1) { next }'
-e syntax OK
>
```

but if you try to run it normally, you'll get an error:

```
>perl -e 'if (1) { next }'
Can't "next" outside a loop block at -e line 1.
>
```

This is the difference between a compile-time and a runtime error. A compile-time error can be detected in advance and means that perl couldn't understand what you said. A runtime error means that what you said was comprehensible but (for whatever reason) can't be done.
`-c` only checks for compile-time errors.

-i

When we're searching and replacing the contents of a file, we usually don't want to produce a new, revised copy on standard output, but rather change the file as it stands. You might think of doing something like this:

```
>perl -pe 's/one/two/g' textfile.txt > textfile.txt
```

There's a problem with that though, as you'll know if you've tried it, it's quite possible that you'll completely lose the file. This is because (unless you're running in a shell that's smart enough to watch your back) the shell opens the file it's writing to first and **then** passes the filehandle to perl as standard output. Perl opens the file after this has taken place, but by this time, the original contents of the file have been wiped out.

To get around this yourself, you'd have to go through contortions like this:

```
>perl -pe 's/one/two/g' textfile.txt > textfile.new
>mv textfile.new textfile.txt
```

The UNIX command `mv` is the same as the `ren` command in Windows: Both commands are used to rename files.

Perl provides you with a way to avoid this. The `-i` switch opens a temporary file and automatically replaces the file to be edited with the temporary file after processing. You can do what we want just like this:

```
>perl -pi -e 's/one/two/g' textfile.txt
```

Well, you *might* be able to – as it stands, you may find that this just returns the message:

```
Can't do inplace edit without backup
```

This happens because perl doesn't know how you want to name the temporary file. Notice though, that I separated `-i` from the `-e` switch: this is because `-i` takes an optional argument. Anything immediately following the `-i` will be treated as an extension to be added to the original filename as a name for the backup file. So, for instance:

```
>perl -pi.old -e 's/one/two/g' textfile.txt
```

will take in a file, `textfile.txt`, save it away as `textfile.txt.old`, and then replace every instance of 'one' with 'two' in `textfile.txt`.

-M

If you need to load any modules from the command line, you can use the `-M` switch. For instance, to produce politically correct one-liners, we should really say something like this:

```
>perl -Mstrict -Mwarnings –e ...
```

However, the kind of code we're likely to put on the command line doesn't really need this sort of strictness. It's still useful to have the -M switch to load modules – the CPAN modules `LWP::Simple`, `Tk`, and `HTML::Parser` have been used in the past to create a one-line graphical web browser!

-s

As well as passing switches to perl, you may want your program to have switches of its own. The `-s` switch (usually specified on the shebang line) tells perl to interpret all command line switches following the filename as variables (for example: `$v`, `$h`) and removed from `@ARGV`. This means that you can process these switches in any way you want.

For instance, a lot of programs will display a help message explaining their usage, when called with the `-h` switch at the command line. Similarly, they'll give their version number if `-v` is used. Let's make some of our own programs do this:

Try It Out : Reading Command Line Options

We're going to add 'help' and 'version number' messages to `nl.plx`, the line numbering program we wrote in the last chapter. Note that this example uses `our` and will therefore only work for Perl versions 5.6 and above:

```
#!/usr/bin/perl -s
# nl3.plx
use warnings;
use strict;

my $lineno;
my $current = "";
our ($v,$h);

if (defined $v) {
   print "$0 - line numberer, version 3\n";
   exit;
}
if (defined $h) {
   print <<EOF;
$0 - Number lines in a file

Usage : $0 [-h|-v] [filename filename...]

This utility prints out each line in a file to standard output,
with line numbers added.
EOF
   exit;
}

while (<>) {
    if ($current ne $ARGV) {
        $current = $ARGV;
        print "\n\t\tFile: $ARGV\n\n";
        $lineno=1;
    }
    print $lineno++;
    print ": $_";
}
```

If we now pass the `-h` option, it's not treated as a filename, but rather as a request for help:

```
>perl –s nl3.plx -h
nl3.plx  Number lines in a file

Usage : nl3.plx [-h|-v] [filename filename...]

This utility prints out each line in a file to standard output,
with line numbers added.
>
```

If you're fortunate enough to be using an operating system that allows you to use Perl programs as executables, the shebang line will take care of specifying the `–s` switch, so you won't need to repeat it on the command line. So while this will work fine on UNIX:

```
>nl3.plx -v
nl3.plx - line numberer, version 3
```

you'll probably need to say this on Windows:

```
>perl –s nl3.plx –v
nl3.plx - line numberer, version 3
```

How It Works

The `-s` on the shebang or command line tells perl that any switches following the name of the Perl program will cause a Perl variable of the same name to be defined. For instance, this command line:

```
>perl -s something -v -abc
```

will set the variables `$v` and `$abc`. We therefore need to be ready to receive these variables, otherwise we will fall foul of `use strict`, so we put:

```
our ($v,$h);
```

If the variable is defined, we do something with it:

```
if (defined $v) {
   print "$0 - line numberer, version 3\n";
   exit;
}
```

The special variable `$0` contains the name of the program currently being run. It's good form to put this in any informational messages you produce about the program.

While `-s` is handy for quick tasks, there are two things that make it unsuitable for use in big programs:

- You have no control over what switches should be recognized. Perl will set any variable, regardless of whether you want it to or not. If you're not actually using that switch, this will generate warnings. For instance:

  ```
  >perl -s nl3.plx -v -foobar
  Name "main::foobar" used only once: possible typo.
  nl3.plx - line numberer, version 3
  >
  ```

- `-abc` is treated as one switch and sets `$abc`, rather than the three switches `-a`, `-b`, and `-c`.

For this reason, it's recommended that you use the standard modules `Getopt::Std` or `Getopt::Long` instead. Appendix D gives a brief rundown of all perl's standard modules or for more detailed information, refer to the `perlmod` manpage.

-I and @INC

Perl knows where to look for modules and Perl files included via `do` or `require` by looking for them in the locations given by a special variable, the array `@INC`. You can add directories to this search path on the command line, by using the -I option:

```
perl –I/private/perl program
```

will cause Perl to look in the directory `/private/perl` for any files it needs to find besides those in `@INC`. For more details on working with `@INC` up close, just have a look in the next chapter.

-a and -F

One of perl's ancestors is the UNIX utility `awk`. The great thing about `awk` was that when reading data in a tabular format, it could automatically split each column into a separate variable for you. The perl equivalent would use an array and would look something like this:

```
while (<>) {
   my @array = split;
   ...
}
```

The `-a` switch, used with `-n` and `-p`, does this kind of `split` for you. It splits to the array `@F`, so

>perl -an '...'

is equivalent to:

```
LINE: while (defined($_ = <ARGV>)) {
    @F = split;
    'Your code here';
}
```

So, to get the first word of every line in a file, you could say this:

>perl -ane 'print $F[0],"\n"' chapter9.txt
Running

By

You

Everyone
…
>

By default, `-a` splits on spaces – although you can change it by specifying another switch, `-F`, which will take your chosen delimiter as an argument. For instance, the fields in a UNIX password file are (as we saw in Chapter 5), delimited by colons. We can extract the home directory from the file by looking at the fifth element of the array. If our `passwd` file contains the line:

```
simon:x:10018:10020::/home/simon:/bin/bash
```

We'll get the following result:

```
>perl -F: -ane 'print $F[5],"\n" if /^simon/' passwd
/home/simon
>
```

-l and –0

It's rather annoying to have to specify `"\n"` on the end of everything we `print`, just to get a new line, especially if we're doing things on the command line. The `-l` switch sets the *output* record separator `$\` equal to the current value of the *input* record separator `$/`. The former is added on automatically at the end of every `print` statement. Since the latter is usually `\n`, the newline character, `-l` adds a newline to everything we print. Additionally, if used with the `-n` or `-p` switches, it will automatically `chomp` any input.

We can cut the above program down by a few more keystrokes like this:

```
>perl -F: -lane 'print $F[5] if /^simon/' passwd
/home/simon
>
```

If `-l` is followed by a valid octal number, then the character with that ASCII value (see Appendix F) is used as the output record separator instead of new line. However, this is relatively rare.

Alternatively, you can set the input record separator using the `-0` switch. Likewise, if this is followed by an octal number, `$/` will be set to that character. For instance, `-0100` will effectively execute `$/="A";` at the beginning of the program. `-0` on its own or followed by something that isn't an octal number will cause `$/` to be set to the `undefined` value, causing the entire file to be read in at once:

> **While you can conceivably use `-l` on the shebang line to save printing newlines in your program, it's actually a bad idea – many people will probably miss it and wonder where all the new lines are coming from. It will also get you into trouble if you want a `print` statement that doesn't cause a newline.**

-T

When you're dealing with data that's being downloaded from an unreliable source from the outside world, you'll probably want to be careful what you do with it. If you're asking the user for a filename, which you pass directly to `open`, you're potentially allowing the user to do all kinds of horrible things. Say, for instance, you were given the filename `rm -rf /|` and used it as it was (**DON'T!**). You may well find afterwards that several of the files on your disk had disappeared...

To force you to clean up this insecure data, Perl has a switch, `-T`, that turns on 'taint mode'. When this switch is in operation, any data coming into your program is tainted, and may not be used for any operations Perl deems 'unsafe' for example, passing to `open`. Furthermore, any data derived from tainted data becomes tainted itself. The **only** way to untaint data is to take a regular expression backreference:

```
$tainted =~ /([\w.]+)/;
$untainted = $1;
```

We'll look at this in a lot more detail in the section on taint checking in Chapter 13.

Debugging Techniques

Earlier in the chapter, we looked at bugs that perl can trap easily – bugs that turn up when what you write doesn't make sense. However, a lot of the time you'll write something that makes sense but doesn't do what you want it to do. While there's no magic formula to find the problem for you here, there are several techniques you can use to track down the problem. Perl itself includes a debugging environment to help you in your investigations.

Before the Debugger...

Before I explain how the debugger works, though, I have to admit that I'm an old-fashioned soul, and don't really believe in debuggers. People seem to see the debugger as a substitute for understanding the problem – just run the program through the debugger, and it'll magically uncover the error. While that would be lovely, it's not actually the case. The debugger can only help you along the way, and there are other ways of debugging a program that may well be far more effective than firing it up.

Debugging Prints

It's an old programming proverb: When in doubt, print it out. Are you sure that the data coming into your program is what you think it is? Print it out! Do you know that a regular expression has done what you think it should have to a variable? Print it out, before and after. Do you know how many times Perl has gone through a certain loop or section of code? Is Perl taking far longer than it should with something trivial? Print out a little message saying where you are in the code. `print` is by far the most powerful and useful debugging tool at your disposal!

Pare It Down

If you're not sure where an error is occurring, try and isolate it. Cut or comment out unrelated lines and see if the problem still occurs. Keep commenting lines out until the problem goes away, and then look at what you've changed.

The same technique can be used when you've got inexplicable behavior. It's a lot easier to spot a bug when the odd behavior is demonstrated in five lines than in fifty. Alternatively, if you can't reproduce the problem that way, start a new program that *just* has the troublesome logic in it and see if you can find anything odd about that. This will also test whether there's something wrong with the data you're feeding into your program.

In any case, the smaller you can make your demonstration code, the better – especially if you're planning on asking someone else about it. The smaller your haystack, the more chance you have of finding a needle in it. Furthermore, plenty of people may be willing to help you if you can produce two lines that demonstrate a problem – fewer will if they think you expect them to debug your entire program.

Here are a some other problems that can cause weirdness without actually causing an error:

Context

Is there a problem with context? Always make sure you know what you're expecting from a function – whether you want an array or a scalar – and ensure that you're collecting the result in the appropriate type of variable.

Scope

Has a variable you've declared with `my` gone out of scope and become undefined or returned to its original value (from before you `my`'d it)? Remember that declaring variables `my` inside a block or loop means you won't be able to get at their value outside of it.

Precedence

Are you saying something like `print (2+3)*5`? This would add two to three, print it, and multiply the result of the `print` by five. Have you forgotten brackets around a list? Having `warnings` turned on will help you pick up on most of these sorts of thing, but be careful. Whenever in doubt, bracket more than you need to.

Using the Debugger

The Perl debugger isn't a separate program, but a special mode under which `perl` runs – to enable it, you simply pass the `-d` switch. As it's a special mode for running your program, you won't get anywhere unless your program compiles correctly first. What the debugger will help you do is to trace the flow of control through your program and allow you to look at variables' values at various stages of operation.

When you start your program in the debugger, you should see something like this:

```
>perl –d nl3.plx
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or `h h' for help, or `perldoc perldebug' for more help.

main::(nl3.plx:6):      my $lineno;
  DB<1>
```

That line '`DB<1>`' is the debugger prompt. Here is a partial list of things you can do at that point:

Command	Description
`T`	Obtain a 'call trace' of all the subroutines perl is currently processing. This will tell you how you got to be where you are.
`s`	Step to the next line as you go one line at a time through your program.

Table continued on following page

Command	Description
`n`	Step over a subroutine. Call the subroutine reference on the current line, and stop again once control has returned from that.
`Return`	Repeat the last stepping command.
`r`	Keep going until the current subroutine returns.
`c`	Continue – keep going until something happens that causes the debugger to stop again.
`l`	List the next few lines to be processed.
`-`	List the previous lines processed.
`w`	List the lines around the current line.
`/pattern/`	Search forwards in the program code until the pattern matches.
`t`	Turn on (or off) trace mode. This prints every statement before executing it.
`b`	Set a breakpoint. Stop running the program and return to the debugger at the given line number or when the given condition is true.
`x`	Evaluate something in array context and give a tree view of the resulting data structure.
`!`	Do the previous command again.
`p`	Print something out.
`h`	Get more help.

We're not going to look any further at the debugger. While it can help you out – and once you start really developing in Perl it really will – for the time being it's a better learning experience to try and debug your code using the hints and techniques shown in the rest of this chapter. That way you can really get to know how Perl thinks and works.

Defensive Programming

Far and away the best way to debug your code is to try and make sure you never have to. While it's impossible to guarantee that there will never be any bugs in your program, there are a lot of things you can do to minimize their number, and to make sure that any potential bugs are easy to locate. In a sense, it's all about expecting the worst.

Strategy

Before writing another line, make sure you've got a plan. You need to use just as methodical an approach to debug code efficiently as you do to write it in the first place. Keep the following points in mind:

- Never try to write a large program without trying parts of that program first. Break the task down into small units, which can be tested as they're written.
- Track down the first bug, then try the program again – the second may just have been a consequence of the first one.

- Likewise, look out for additional errors after you've 'fixed' the first bug. There could have been a knock-on effect revealing subsequent errors.

Check Your Return Values

There's no excuse for not checking the return values on any operator that gives a meaningful return value. Any operator that interacts with the system will return something by which you can determine whether it succeeded or not, so make use of it. Furthermore, you can always attempt to pre-empt problems by looking to see what could go wrong. Chapter 6 contained an example of defensive programming, when we tested whether files were readable and writeable.

Be Prepared for the Impossible

Sometimes, things don't go the way you think they should. Data can get shuffled or wiped out by pieces of code in ways that you can't explain. In order to pick this up as soon as possible after it happens, test to see if the impossible has occurred. If you know a number's going to be 1, 2, or 3, do something like this:

```
if ($var == 1) {
    # Do first thing.
} elsif ($var == 2) {
    # Do second thing.
} elsif ($var == 3) {
    # Do third thing.
} else {
    die "Whoa! This can't happen!\n";
}
```

With luck, you'll never get there, but if you do, you'll be alerted to the fact that something higher up in the program has wiped out the variable. These are a type of trap called **assertions** or, less formally, '**can't happen' errors**. Eric Raymond, author of 'The Jargon File', says this about them:

> *"Although 'can't happen' events are genuinely infrequent in production code, programmers wise enough to check for them habitually are often surprised at how frequently they are triggered during development and how many headaches checking for them turns out to head off."*

Never Trust the User

Users are an extremely reliable source of bad data. Don't let bad data be the cause of bugs. Check to ensure you're getting the sort of data you want. Do you want to take the newline character off the end? Are you expecting to be upper case, lower case, mixed case, or don't you care? Try and be flexible wherever possible, since the user is more than likely to get something wrong. Above all, make sure you're completely happy with input before acting on it.

Definedness and Existence

If you're putting elements into arrays or hashes, should they already exist? Should they not exist? Check that you're not wiping data you want to keep, and if you are, ask yourself how you got into that situation. Are you sure you've got some data to put in? Check that you're putting the right sort of data into the right place. Are you sure there's something there when you take data out? Make sure the data exists when you've accessed a hash or array.

Have Truthful, Helpful Comments

Comments are a useful memory aid to help you keep track of what's going on in the program, so try and use them as intended. Comments that explain data flow – what the data means and where it comes from – are more helpful than comments that explain what you're doing. Contrast the usefulness of these two sections:

```
$a = 6.28318; # Assign 6.28318 to $a


$a = 6.28318; # pi*2
```

The problem with comments is that you have to keep them up to date when you change the code. Make sure your comments aren't a distraction (at best) or (at worst) downright misleading. There's an old saying: 'If code and comments disagree, both are probably wrong.'

Keep the Code Clean

Tidy code is much easier to understand and debug than messy code. It's easier to find problems if, among other things, you make sure to always:

- keep parallel items aligned together in columns
- keep indentation regular
- keep to one statement per line
- split long statements over multiple lines
- use white-space characters to increase readability

Again, contrast these two snippets. There's this:

```
while (<>) {
   if ( /^From:\s+(.*)/    ) { $from    = $1 }
   if ( /^Subject:\s+(.*)/ ) { $subject = $1 }
   if ( /^Date:\s+(.*)/    ) { $date    = $1 }


   print "Mail from $from on $date concerning $subject\n"
      unless /\S+/;

   next until /^From/;
}
```

versus this:

```
while(<>){if(/^From:\s+(.*)/){$from=$1}
if(/^Subject:\s+(.*)/){$subject=$1}
if(/^Date:\s+(.*)/){$date=$1}
print "Mail from $from on $date concerning $subject\n" unless /\S+/;
next until /^From/;}
```

Which one would *you* rather debug?

Summary

Whenever you program, you'll inevitably make mistakes and create bugs. There are two types of bug you'll come up against: the syntax error, which manifests itself with a violent bang, and the logic error, which hides away insidiously inside your program and drives you silently mad. This chapter has shown you how you can deal with both sorts of bug.

We've looked at Perl's error messages and the most common causes of syntax errors. We've seen how to decode the error messages perl gives, both by employing a little bit of logical thought (the best way) and by getting the `diagnostics` pragma to explain it to us (the easiest way).

We've also seen how to avoid creating bugs in the first place – `use warnings` and `use strict` act as checks to ensure that we're not doing anything too crazy. There are also plenty of ways to use defensive programming, imposing further checks to stop bugs before they happen.

Perl is a great tool for use on the command line. I'm forever using it to search files for patterns and change files with a search-and-replace, as well as using it to test out snippets of Perl code and examine Perl's behavior. We've looked at various command line switches, which make it easy for us to do complex things: loop over a file, change a file in place, check the syntax of a file or piece of code, and so on.

We've also doffed our cap to the Perl debugger, as well as some other ways to detect and remove bugs in our code. Now you're armed to do battle with any bugs that come your way – and come they will!

Exercises

Take a look at the following file, apply what you've read about, and see if you can knock it into shape:

```
#!/usr/bin/perl
#buggy.plx

my %hash;

until (/^q/i) {

print "What would you like to do? ('o' for options): "
$  = STDIN;

if ($  eq "o"){options}elsif($  eq "r"){read}elsif($  eq "l"){ list }elsif
($  eq "w"){ write }elsif ($  eq "d") { delete } elsif ($  eq "x") { clear }
else { print "Sorry, not a recognized option.\n"; }

sub options {
   print<<E0F
      Options available:
      o - view options
      r - read entry
      l - list all entries
      w - write entry
```

```
      d - delete entry
      x - delete all entries
EOF;
}

sub read {
my $keyname = getkey();

if (exists $hash{"$keyname"}) {
print "Element '$keyname' has value $hash{$keyname}";
} else {
print "Sorry, this element does not exist.\n"}}

sub list {foreach (sort keys(%hash)) {print "$  => $hash{$ }\n";}}

sub write {
my $keyname = getkey();
my $keyval = getval();
if (exists $hash{$keyname}) {print "Sorry, this element already exists.\n"
} else {$hash{$keyname}=$keyval;}}

sub delete {
my $keyname = getkey();
if (exists $hash{$keyname}) {
print "This will delete the entry $keyname.\n";
delete $hash{$keyname};}}

sub clear {undef %hash;}

sub getkey {print "Enter key name of element: "; chomp($  = <STDIN>);)

sub getval {print "Enter value of element: "; chomp($_ = <STDIN>);}
```