6

Files and Data

We're starting to write real programs now, and real programs need to be able to read and write files to and from your hard drive. At the moment, all we can do is ask the user for input using `<STDIN>` and print data on the screen using `print`. Pretty simple stuff, yes, but these two ideas actually form the basis of a great deal of the file handling you'll be doing in Perl.

What we want to do in this chapter is extend these techniques into reading from and writing to files, and we'll also look at the other techniques we have for handling files, directories, and data.

Filehandles

First though, let's do some groundwork. When we're dealing with files, we need something that tells Perl which file we're talking about, which allows us to refer to and access a certain file on the disk. We need a label, something that will give us a 'handle' on the file we want to work on. For this reason, the 'something' we want is known as a **filehandle**.

We've actually already seen a filehandle: the **`STDIN`** of `<STDIN>`. This is a filehandle for the special file 'standard input', and whenever we've used the idiom `<STDIN>` to read a line, we've been reading from the standard input file. Standard input is the input provided by a user either directly as we've seen, by typing on the keyboard, or indirectly, through the use of a 'pipe' that (as we'll see) pumps input into the program.

As a counterpart to standard input, there's also standard output: **`STDOUT`**. Conversely, it's the output we provide to a user, which at the moment we're doing by writing to the screen. Every time we've used the function print so far, we've been implicitly using `STDOUT`:

```
print STDOUT "Hello, world.\n";
```

is just the same as our original example in Chapter 1. There's one more 'standard' filehandle: standard error, or **`STDERR`**, which is where we write the error messages when we `die`.

Every program has these three filehandles available, at least at the beginning of the program. To read and write from other files, though, you'll want to open a filehandle of your own. Filehandles are usually one-way: You can't write to the user's keyboard, for instance, or read from his or her screen. Instead, filehandles are open either for reading or for writing, for input or for output. So, here's how you'd open a filehandle for reading:

```
open FH, $filename or die $!;
```

The operator for opening a filehandle is `open`, and it takes two arguments, the first being the name of the filehandle we want to open. Filehandles are slightly different from ordinary variables, and they do not need to be declared with `my`, even if you're using `strict` as you should. It's traditional to use all-capitals for a filehandle to distinguish them from keywords.

The second argument is the file's name – either as a variable, as shown above, or as a string literal, like this:

```
open FH, 'output.log' or die $!;
```

You may specify a full path to a file, but don't forget that if you're on Windows, a backslash in a double-quoted string introduces an escape character. So, for instance, you should say this:

```
open FH, 'c:/test/news.txt' or die $!;
```

rather than:

```
open FH, "c:\test\news.txt" or die $!;
```

as `\t` in a double-quoted string is a tab, and `\n` is a new line. You could also say `"c:\\test\\news.txt"` but that's a little unwieldy. My advice is to make use of the fact that Windows allows forward slashes internally, and forward slashes do not need to be escaped: `"c:/test/news.txt"` should work perfectly fine.

So now we have our filehandle open – or have we? As I mentioned in Chapter 4, the `X or Y` style of conditional is often used for ensuring that operations were successful. Here is the first real example of this.

When you're dealing with something like the file system, it's dangerous to blindly assume that everything you are going to do will succeed. A file may not be present when you expect it to be, a file name you are given may turn out to be a directory, something else may be using the file at the time, and so on. For this reason, you need to check that the `open` did actually succeed. If it didn't, we `die`, and our message is whatever is held in `$!`.

What's `$!`? This is one of Perl's **special variables**, designed to give you a way of getting at various things that Perl wants to tell you. In this case, Perl is passing on an error message from the system, and this error message should tell you why the `open` failed: It's usually something like 'No such file or directory' or 'permission denied'.

There are special variables to tell you what version of Perl you are running, what user you are logged in as on a multi-user system, and so on. Appendix B contains a complete description of Perl's special variables.

So, for instance, if we try and open a file that is actually a directory, this happens:

```
#!/usr/bin/perl
# badopen.plx
use warnings;
use strict;
open BAD, "/temp" or die "We have a problem: $!";
```

```
>perl badopen.plx
Name "main::BAD" used only once: possible typo at badopen.plx line 5
We have a problem: Permission denied at badopen.plx line 5.
>
```

The first line we see is a warning. If we were to finish the program, adding further operations on `BAD` *(or get rid of* `use warnings`*), it wouldn't show up.*

You should also note that if the argument you give to `die` does not end with a new line, Perl automatically adds the name of the program and the location that had the problem. If you want to avoid this, always remember to put new lines on the end of everything you `die` with.

Reading Lines

Now that we can open a file, we can then move on to reading the file one line at a time. We do this by replacing the `STDIN` filehandle in `<STDIN>` with our new filehandle, to get `<FH>`. Just as `<STDIN>` reads a single line from the keyboard, `<FH>` reads one line from a filehandle. This <…> construction is called the **diamond operator**, or **readline operator**:

Try It Out : Numbering Lines

We'll use the `<FH>` construct in conjunction with a `while` loop to go through each line in a file. So then, to print a file with line numbers added, you can say something like this:

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

open FILE, "nlexample.txt" or die $!;
my $lineno = 1;

while (<FILE>) {
   print $lineno++;
   print ": $_";
}
```

Now, create the file `nlexample.txt` with the following contents:

```
One day you're going to have to face
   A deep dark truthful mirror,
And it's gonna tell you things that I still
   Love you too much to say.
####### Elvis Costello, Spike, 1988 #######
```

This is what you should see when you run the program:

```
>perl nl.plx
1: One day you're going to have to face
2:     A deep dark truthful mirror,
3: And it's gonna tell you things that I still
4:     Love you too much to say.
5. ####### Elvis Costello, Spike, 1988 #######
>
```

How It Works

We begin by opening our file and making sure it was opened correctly:

```
open FILE, "nlexample.txt" or die $!;
```

Since we're expecting our line numbers to start at one, we'll initialize our counter:

```
my $lineno = 1;
```

Now we read each line from the file in turn, which we do with a little magic:

```
while (<FILE>) {
```

This syntax is actually equivalent to:

```
while (defined ($_ = <FILE>)) {
```

That is, we read a line from a file and assign it to `$_`, and we see whether it is defined. If it is, we do whatever's in the loop. If not, we are probably at the end of the file so we need to come out of the loop. This gives us a nice, easy way of setting `$_` to each line in turn.

As we have a new line, we print out its line number and advance the counter:

```
    print $lineno++;
```

Finally, we print out the line in question:

```
    print ": $_";
```

There's no need to add a newline since we didn't bother `chomp`ing the incoming line. Of course, using a statement modifier, we can make this even more concise:

```
open FILE, "nlexample.txt" or die $!;
my $lineno = 1;

print $lineno++, ": $_" while <FILE>
```

But since we're going to want to expand the capabilities of our program -adding more operations to the body of the loop – we're probably better off with the original format.

Creating Filters

As well as the three standard filehandles, Perl provides a special filehandle called **ARGV**. This reads the names of files from the command line and opens them all, or if there is nothing specified on the command line, it reads from standard input. Actually, the @ARGV array holds any text after the program's name on the command line, and <ARGV> takes each file in turn. This is often used to create filters, which read in data from one or more files, process it, and produce output on STDOUT.

Because it is used so commonly, Perl provides an abbreviation for <ARGV>: an empty diamond, or <>. We can make our line counter a little more flexible by using this filehandle:

```
#!/usr/bin/perl
# nl2.plx
use warnings;
use strict;

my $lineno = 1;

while (<>) {
   print $lineno++;
   print ": $_";
}
```

Now Perl expects us to give the name of the file on the command line:

```
>perl nl2.plx nlexample.txt
1: One day you're going to have to face
2:     A deep dark truthful mirror,
3: And it's gonna tell you things that I still
4:     Love you too much to say.
5. ####### Elvis Costello, Spike, 1988 #######
>
```

We can actually place a fair number of files on the command line, and they'll all be processed together. For example:

```
>perl nl2.plx nlexample.txt nl2.plx
1: One day you're going to have to face
2:     A deep dark truthful mirror,
3: And it's gonna tell you things that I still
4:     Love you too much to say.
5. ####### Elvis Costello, Spike, 1988 #######
6: #!/usr/bin/perl
7: # nl2.plx
8: use warnings;
9: use strict;
10:
11: my $lineno = 1;
12:
13: while (<>) {
14:     print $lineno++;
15:     print ": $_";
16: }
```

If we need to find out the name of the file we're currently reading, it's stored in the special variable $ARGV. We can use this to reset the counter when the file changes.

Try it out : Numbering Lines in Multiple Files

By detecting when $ARGV changes, we can reset the counter and display the name of the new file:

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

my $lineno;
my $current = "";

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

   print $lineno++;
   print ": $_";
}
```

And now we can run this on our example file and itself:

>perl nl3.plx nlexample.txt nl3.plx

```
                File: nlexample.txt

1: One day you're going to have to face
2:      A deep dark truthful mirror,
3: And it's gonna tell you things that I still
4:      Love you too much to say.
5. ####### Elvis Costello, Spike, 1988 #######

                File: nl3.plx

1: #!/usr/bin/perl
2: #nl3.plx
3: use warnings;
4: use strict;
5:
6: my $lineno;
7: my $current ="";
8:
9: while (<>) {
10:      if ($current ne $ARGV) {
11:           $current =$ARGV;
12:           print "\n\t\tFile: $ARGV\n\n";
13:           $lineno=1;
14:      }
15:
16:      print $lineno++;
17:      print ": $_";
18: }
>
```

How It Works

This is a technique you'll often see in programming to detect when a variable has changed. `$current` is meant to contain the current value of `$ARGV`. But if it doesn't, `$ARGV` has changed:

```
if ($current ne $ARGV) {
```

so we set `$current` to what it should be – the new value – so we can catch it again next time:

```
   $current = $ARGV;
```

We then print out the name of the new file, offset by new lines and tabs:

```
   print "\n\t\tFile: $ARGV\n\n";
```

and reset the counter so we start counting the new file from line one again.

```
   $lineno=1;
}
```

As with most tricks like these, it's actually really simple to code it once you've seen how it's coded. The catch is having to solve problems like these for the first time by yourself.

Reading More than One Line

Sometimes we'll want to read more than one line at once. When you use the diamond operator in a scalar context, as we've been doing so far, it'll provide you with the next line. However, in a list context, it will return all of the remaining lines. For instance, you can read in an entire file like this:

```
open INPUT, "somefile.dat" or die $!;
my @data;
@data = <INPUT>;
```

This is, however, quite memory-intensive. Perl has to store every single line of the file into the array, whereas you may only want to be dealing with one or two of them. Usually, you'll want to step over a file with a `while` loop as before. However, for some things, an array is the easiest way of doing things. For example, how do you print the last five lines in a file?

The problem with reading a line at a time is that you don't know how much text left you've got to read. You can only tell when you run out of data, so you'd have to keep an array of the last five lines read and drop an old line when a new one comes in. You'd do it something like this:

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

open FILE, "gettysburg.txt" or die $!;
my @last5;

while (<FILE>) {
   push @last5, $_; # Add to the end
   shift @last5 if @last5 > 5; # Take from the beginning
}

print "Last five lines:\n", @last5;
```

And that's exactly how you'd do it if you were concerned about memory use on big files. Given a suitably primed `gettysburg.txt`, this is what you'd get:

```
>perl tail.plx
Last five lines:
- that from these honored dead we take increased devotion to that cause for
which they gave the last full measure of devotion - that we here highly resolve
that these dead shall not have died in vain, that this nation under God shall
have a new birth of freedom, and that government of the people, by the people,
for the people shall not perish from the earth.
>
```

However, if memory wasn't a problem, or you knew you were going to be primarily dealing with small files, this would be perfectly sufficient:

```
#!/usr/bin/perl
# tail2.plx
use warnings;
use strict;

open FILE, "gettysburg.txt" or die $!;
my @speech = <FILE>;

print "Last five lines:\n", @speech[-5 ... -1];
```

What's My Line (Separator)?

So far we've been reading in single lines – a series of characters ending in a new line. One of the other things we can do is to alter Perl's definition of what separates a line.

The special variable `$/` is called the 'input record separator'. Usually, it's set to be the newline character, `\n`, and each 'record' is a line. We might say more correctly that `<FILE>` reads a single **record** from the file. Furthermore, `chomp` doesn't just remove a trailing new line – it removes a trailing record separator. However, we can set this separator to whatever we want, and this will change the way Perl sees lines. So if, for instance, our data was defined in terms of paragraphs, rather than lines, we could read one paragraph at a time by changing `$/`.

Try It Out : Fortune Cookie Dispenser

The fortune cookies file for the UNIX `fortune` program – as well as some 'tagline' generators for e-mail and news articles – consist of paragraphs separated by a percent sign on a line of its own, like this:

```
We all agree on the necessity of compromise.  We just can't agree on
when it's necessary to compromise.
    -- Larry Wall
%
All language designers are arrogant.  Goes with the territory...
    -- Larry Wall
%
Oh, get a hold of yourself. Nobody's proposing that we parse English.
    -- Larry Wall
%
Now I'm being shot at from both sides. That means I *must* be right.
     -- Larry Wall
%
```