9 Reading from a file

In general, a file you want to get data into your program from will not necessarily be in the same directory as the program itself; it may have to be located by a pathname which could be long and complicated. The structure of pathnames differs between operating systems; if you are working in a Unix environment, for instance, the pathname might be something like:

```
../jjs/weather/annualRecords.txt
```

Whatever pathnames look like in your computing environment, to read data into a Perl program you have to begin by defining a convenient *handle* which the program will use to stand for that pathname. For instance, if your program will be using only one input file, you might choose the handle `INFILE` (it is usual to use capitals for filehandles).

The code:

```
open(INFILE, "../jjs/weather/annualRecords.txt");
```

says that, from now until we hit a line `close(INFILE)`, any reference to `INFILE` in the program will be reading in data from the annualRecords file specified in the pathname.

Having "opened" a file for input, we use the symbol `<>` to actually read a line in. Thus:

```
$a = <INFILE>;
```

will read in a line from the annualRecords file and assign that string of characters as the value of `$a`.

A line from a multi-line file will terminate in one or more line-end characters, and the identity of these may depend on the system which created the file (different operating systems use different line-end characters). Commonly, before doing anything else with the line we will want to convert it into an ordinary string by removing the line-end characters, and the built-in function `chomp()` does that. This is an example of a function whose main purpose is to change its argument rather than to return a value; `chomp()` does in fact return a value, namely the number of line-end characters found and removed, but programs will often ignore that value – they will say e.g. `chomp($line)`, rather than saying e.g. `$n = chomp($line)`, with follow-up code using the value of `$n`.

(If no filehandle is specified, `$a = <>` will read in from the keyboard – the program will wait for the user to type a sequence of characters ending in a newline, and will assign that sequence to `$a`.[11])

Assuming that we are reading data from a file rather than from the keyboard, what we often want to do is to read in the *whole* of the input file, line by line, doing something or other with each successive line. An easy way to achieve that is like this:

```
while ($a = <INFILE>)
  {
  chomp($a);
  do something with $a
  }
```

The word `while` tests for the truth of a condition; in this case, it tests whether the assignment statement, and hence the expression `<INFILE>`, is true or false. So long as lines are being read in from the input file, `<INFILE>` counts as "true", but when the file is exhausted `<INFILE>` will give the value "false". Hence `while ($a = <INFILE>)` assigns each line of the input file in turn to `$a`, and ceases reading when there is nothing more to read. (It is a good idea then to include an explicit `close(INFILE)` statement, though that is not strictly necessary.)

Our `open ...` statement assumed that the annualRecords file was waiting ready to be opened at the place identified by the pathname. But, of course, that kind of assumption is liable to be confounded! Even supposing we copied the pathname accurately when we typed out the program, if that was a while ago then perhaps the annualRecords file has subsequently been moved, or even deleted. In practice it is virtually mandatory, whenever we try to open a file, to provide for the possibility that it does not get opened – normally, by using a `die` statement, which causes the program to terminate after printing a message about the problem encountered. A good way to code the `open` statement will be:

```
open(INFILE, "../jjs/weather/annualRecords.txt") or
   die("Can\'t open annualRecords.txt\n");
```

Between actions, as here, the word `or` amounts to saying "Do the action on the left if you can, but if you can't, then do the action on the right".

Sometimes we may want to deal with input one character at a time, rather than a whole line at a time, and Perl does have ways of reading in single characters. But these techniques involve system-dependent complications. When one is new to Perl, it is best to read in complete lines, and then break the lines up into separate characters and deal with them individually within one's program. (Chapter 12 will show us an easy way of breaking a line into a set of characters.)

The above tells us how to read data in from a file. The converse operation, writing data from our program to an external file, will be covered in chapter 11 below.

Since we have looked at `die`, which terminates a program after displaying a message and is commonly used to catch errors (such as files not being located where they are expected to be), we should end this chapter with a discussion of other ways in which Perl programs can terminate.

The most straightforward is that the flow of control simply runs out of code. Our very first program (1) executed three statements in sequence; there was nothing left to execute, so the program terminated. Often, though, we shall want to make the termination point explicit. We may want the program to terminate long before reaching the last line of code, if some condition is met.

The keyword for this is `exit`. We'll illustrate the use of this by example.

We have not yet discussed what the annualRecords file contains, but let's suppose that it comprises a record for each year since 1900, containing statistics of rainfall and average high and low temperatures, in a format which begins with the year number, like this:

```
        ⋮
1949    1023 mm    13.7 degC  6.0 degC
1950     876 mm    14.2 degC  7.1 degC
        ⋮
```

Let's say that we want to extract and print out just the records for the 1970s. Here is a program which will do that:

(8)

```
1   open(INFILE, "../jjs/weather/annualRecords.txt") or
    die("Can\'t open weather data file\n");
2   while ($line = <INFILE>)
3     {
4     chomp($line);
5     $date = substr($line, 0, 4);
6     if ($date < 1970)
7       {;}
8     elsif ($date > 1979)
9       {
10      close(INFILE);
11      exit;
12      }
13    else
14      {
15      print "$line\n";
16      }
17    }
```

The `if - elsif - else` construction begins by checking for years outside the period of interest. If the date shows that the 1970s have not yet been reached, we want nothing at all to be done with that input line, as shown by a block containing just a semicolon not preceded by any statement on line 8.7. The current pass through the `while` loop will end, and the next line will be read in. (Normally we are writing blocks with the opening and closing curly brackets, and whatever comes between them, on separate lines; but we know that Perl does not care about that, so with just three characters in the block it seemed simplest to put the whole block on one line.) If the program has reached a year beyond the 1970s it closes INFILE and terminates via 8.11, without troubling to read in later records. Just in those cases where the program gets as far as the `else` clause can it be dealing with a year in the 1970s, and in those cases `$line` is printed out. The program will never terminate by "falling off the end of the code", as program (1) did: after the input file has been successfully opened in 8.1, the line 8.11 is the only route to program termination.

Strictly, line 8.10 is unnecessary: any files opened by a program are automatically closed when that program terminates. But it is probably good discipline to include explicit `close` statements – later, you might incorporate your simple early program into a larger program which goes on to do other things, in which case it could prove a nuisance if files that are finished with have never been closed.

You might wonder whether it is necessary to `chomp()` the line-end characters off the lines read in (see 8.4), when the only lines that are used will be printed out with a line-end character (`\n`) added (8.15) – isn't this like "marching up to the top of the hill and marching down again"? But `\n` is *your* line-end character; if the weather records file was created by other people working in other computing environments, it may use different conventions. It is wise to make sure that output you generate conforms to your own conventions.

A further point to notice here is that the value assigned to `$date` is created (in 8.5) as a substring of a longer string of characters, which contains letters as well as numbers. But each character of the substring `$date` is a digit – `$date` looks like a number, so we can treat it as a number. In this case we compare `$date` to other numbers, but equally (if we wanted to) we could use `$date` in arithmetic operations such as addition or division. In many programming languages one could not do that. In those languages, a string of digit characters is a character-string, not a number, and we would need to convert it into the number it looks like before we could use it as a number. Perl is more easygoing.

A final point about (8) has to do with *error-trapping*. In connexion with `die` we saw that it is wise not to make too many assumptions about external files being as they ideally should be. Even if the annualRecords file is at the location identified in 8.1 (so that the `die` instruction is not activated), there could easily be unpleasant surprises within its contents. Program (8) is written on the assumption that annualRecords contains a line for each year, that the year name occupies the first four bytes of its line, and that the years are in the correct order. But what if, some year, annualRecords were updated incorrectly, perhaps with the year name at the end rather than the beginning of the line?

At the very least, it would be wise in practice not to take for granted that a line which fails the tests in 8.6 and 8.8 must be a line beginning with a year in the 1970s. We could build in an explicit check, by replacing 8.13–16 with:

```
elsif($date >= 1970 and $date <= 1979)
  {
  print "$line\n";
  }
else
  {
  die("Ill-formed line: $line\n");
  }
```

Perhaps the input file is fine and this `die` instruction will never be triggered, but it costs nothing to include it.

Programs written for real-life purposes tend to contain a great deal of error-trapping – sometimes there will be more error-trapping code than code which we want to be executed. It would be confusing for code examples in a textbook to contain a realistic amount of error-trapping code, because it would distract the reader's attention from the central point being made by a particular example; so the code displayed in this book will often assume that external files are as they should be. But bear in mind when programming in practice that it is wise to think about what *might* go wrong, and to include code to handle it explicitly just in case it does go wrong.