11 Writing to a file

11.1 Reading, writing, appending

Now let's do something a little more ambitious with our file of county populations. We will write a Perl routine to read it in line by line, and write out a sister file in which lines contain the population figure first, expressed as numbers of individuals rather than in thousands, and with commas used conventionally to group digits into threes. In multi-word county names, the words will be separated by underlines rather than spaces, to make them easily recognizable as units in later computer processing. In other words, the fourth and fifth lines of the output file will be in the form:

```
294,000     Cambridgeshire_and_Isle_of_Ely
1,472,000   Cheshire
```

We will make no assumptions about how whitespace is used in the input file; in the output file there will be no leading or trailing whitespace, and populations will be separated from county names by a single tab. Any wholly-blank lines in the input will be eliminated, so the new file will contain one line per county.

This will show us how Perl programs create and write output to external files. At the same time it will give us an opportunity to deepen our understanding of pattern matching (which really is the central topic in Perl programming).

Earlier, we saw how to open a file for reading, giving it a "handle" by which it is referred to in subsequent code. Now, we need also to create a new file and open it for writing, with a filehandle of its own. To specify that we shall be writing to a file rather than reading from it, we put the symbol `>` at the beginning of the pathname within the `open` statement:

```
open(OUTFILE, ">../data/pops/reformedData.txt");
```

A third possibility, not relevant here, is that we are opening an existing file in order to *append* to it – previous file contents are left in place and new material added after them. The symbol for this is `>>`. (The single `>` implies that the file is new, so if it is used with an existing filename that file is liable to be overwritten.)

The symbol `<` is used to say explicitly that a file is being opened for reading; but reading is the default option, so if `<` is omitted (as it was when we introduced the `open` keyword in chapter 9), reading rather than writing or appending is assumed.

Here is a piece of code that will do the task outlined. Most of the Perl constructions used are already familiar, but there are some new points:

(12)

```
1  open(INFILE, "<../data/pops/countyData.txt")
     or die "Cannot open input file\n";
2  open(OUTFILE, ">../data/pops/reformedData.txt")
     or die "Cannot open output file\n";
3  while ($a = <INFILE>)
4    {
5    if ($a !~ /\S/) {;}
6    elsif ($a !~ /^\s*(\S.*\S)\s+(\d+)\s*$/)
7      {
8      die "bad input line:  $a\n";
9      }
10   else
11     {
12     $name = $1;
13     $population = $2;
14     $name =~ s/(\S)\s+(\S)/$1_$2/g;
15     $population =~ s/$/,000/;
16     $population =~ s/(\d)(\d{3},)/$1,$2/;
17     print OUTFILE "$population\t$name\n";
18     }
19   }
20 close(INFILE);
21 close(OUTFILE);
```

We have already discussed lines 1–2 (this time round, I have included the `<` in line 1 for explicitness). Lines 3, 4, and 19, as before, set up a `while` loop to process successive lines from INFILE.

Line 12.5 is included to handle input lines which are wholly blank: they may include various whitespace characters, but they nowhere contain a `\S` (black) character. What we want Perl to do with a line like that is nothing – as indicated by a block containing just a semicolon.

Assuming that the line read in does contain some black characters, we pass to line 6 which checks that its last patch of black characters are all digits, separated by whitespace from the black characters earlier in the line. If this match fails, the program prints an error message and dies. But even though 12.6 asks whether `$a` "fails to match" (`!~`) the pattern, if `$a` *does* match then the two patches of black characters are "captured" by the brackets, so that they are temporarily named `$1` and `$2`. In 12.12–13 these substrings are assigned to the meaningful variable names `$name` and `$population` respectively.

Line 12.14 changes sequences of one or more whitespace characters that are surrounded on both sides by black characters within `$name` into single underline characters. (The pattern of 12.14 will cover only part of a `$name` string – just the space between words and the two characters to its immediate left and right. When the substitution is made, the earlier and later parts of `$name`, outside the part covered by the pattern, will automatically be carried over to the new string – this does not need to be said explicitly.)

The `g` following the `s/.../.../` construction in 12.14 means that this substitution is to apply *globally*: the pattern will repeatedly be matched and the replacement made, until there is nowhere left in the string where the pattern applies. (Without this `g`, OUTFILE would contain county names such as `Cambridgeshire_and Isle of Ely`.)

Because line 12.14 involves a pattern-matching operation with two pairs of round brackets in the pattern section, it gives `$1` and `$2` new values. Those variables were earlier given values in 12.6, and they keep the same values until a subsequent pattern-matching operation changes them. In practice, whenever pattern-matching uses brackets to "capture" substrings, even if we do not plan to do anything with those substrings in the immediate future it is good to give them meaningful names quickly (as in 12.12–13 here), for fear that we might inadvertently change the values of `$1`, etc. with another pattern-matching statement before getting round to using those values.

Line 12.15 modifies the `$population` string by adding `,000` to the end. This doesn't have to be done via pattern-matching; it could equally well be achieved via the concatenation operator:

```
$population .= ",000";
```

I chose to use the substitution construction simply because we have been discussing pattern matching, so I appended material by looking for the pattern "end of string". But notice that `$` for "end of string" is meaningful only within the *pattern* section of an `s/.../.../` construction. We cannot use it in the replacement section; the end of a string is not a separate item which is inserted as part of a replacement, it is a position which *results* from making a replacement. If we had written the replacement section as `/,000$/`, Perl would have given an error message.

If `$population` is now a number of at least seven figures, a further comma needs to be inserted before the sixth digit from the end; 12.16 looks for the pattern "digit – three digits – comma" and inserts a comma after the first digit.

Line 12.17 prints the line to the newly-created output file in the desired format; when a print statement contains a filehandle before the material to be "printed", it is sent to the relevant file rather than displayed on the screen.

Finally, after the `while` loop has been traversed once for each line of the input file, 12.20–21 tidy things up by explicitly closing the input and output files. (If later code is to read or append to either file, a new `open` statement will be needed.)

11.2 Pattern-matching modifier letters

Line 12.14 introduced the symbol `g` for "global", which as we saw causes a match to occur repeatedly at each place where the target string contains the pattern specified. This symbol follows the last slash in either an `s/.../.../` or an `m/.../` construction.

It is obvious what "global substitution" means – make the substitution at each point where the pattern occurs; but at first sight you might wonder what reason there could be to append `g` to an `m/.../` statement. An `m/.../` statement yields the value true or false, and a pattern only needs to occur at one place in a target string for the statement to be true; so what difference does it make if the pattern occurs more than once?

It can make a large difference, though, if the pattern-match statement is within a loop. Consider:

```
$a = "the man in the ice";
while($a =~ m/the (\w*)/g)
  {
  print "$1\n";
  }
```

```
man
ice
```

With `g`, after the first *the* is matched Perl moves on to look for a later *the*, so the successive values of `$1` are the words following the two *the*'s. Without `g`, the `while` statement would repeatedly succeed by matching the first *the*, and would enter an infinite loop, printing out:

```
man
man
man
man
⋮
```

over and over again, until the user forcibly terminates the loop by entering whatever key combination is used to interrupt a process on his system.[16]

The letter `g` for "global" is only one of various letters that can be suffixed to an `s/.../.../` or `m/.../` construction to modify its meaning. Another is `i`, meaning "ignore case of letters":

```
$a = "The Ice Age is over";
if ($a =~ m/ice age/i)
  {
  print("ice age present\n");
  }
```

```
ice age present
```

– the match succeeds, although the pattern has `ice age` and the target string has `Ice Age`.

The modifier letter `x` allows complicated patterns to be set out in a more human-friendly fashion, including whitespace and comments, which are ignored when Perl matches the pattern. So for instance line 11.2 in chapter 10, which contained the complicated pattern match that picked out the word *Toulouse* as containing the same pair of vowels at two places, could alternatively be written as:

```
if ($towns =~ /
   (\s|^)    #whitespace or start of string, followed by:
   (\S*      #any black characters, before:
   ([aeiou][aeiou])     #a pair of vowels
   \S*       #followed by any black characters, before
   \3        #the same pair of vowels
   \S*)      #followed by any black characters
   (\s|$)    #till whitespace or end of string
   /x)
```

– the `x` in the last line here makes these nine lines the equivalent of the single line 11.2. (However, `x` would not help us to write a complicated character class such as `[^\sa-z\/\\\^]` more readably – it does not affect the interpretation of characters within a character class surrounded by square brackets.)

Modifier letters can be combined (in any order, it makes no difference):

```
$a = "The man in the ice";
$a =~ s/the/that/ig;
print "$a\n";
```

that man in that ice

Other pattern-matching modifier letters are too specialized to cover here.

11.3 Generalizing special cases

Returning to our program (12), for reformatting the county population data: although it achieved everything we wanted it to do with the file of Figure 1, it is not really satisfactory as it stands.

Line 12.15 put numbers into a human-friendly format by supplying a comma between the thousands and hundreds digits, and 12.16 extended that by supplying a comma after the millions digit for numbers in the millions. However, using commas to group digits into threes is a general process. Suppose some county had a population in billions; in the output of (12) the number would appear as, say:

```
12546,957,000
```

– which arguably looks odder than it would look with no commas at all.

Of course, it is absurd to imagine that a single English county might have a population in billions. But it is a bad idea to rely on a consideration like that as an excuse for treating a general process as if it were a limited set of special cases. Once we have program (12) running satisfactorily with the county population data, some time later we might want to adapt it to handle, say, property-tax bases (the total values of properties liable to council tax), where in the 21st century the figure for a county certainly would get into billions of pounds. Then we will get an unexpected (and unwelcome) surprise when numbers looking like `12546,957,000` show up in the output.

The “right way” to deal with commas in (12) is to use a single process to insert commas in numbers wherever they are needed: after the thousands, after the millions, and after the billions and indeed trillions if such large numbers ever arise. Line 12.15 should be changed so that it only adds the necessary zeros, without adding a comma:

```
15  $population =~ s/$/000/;
```

and 12.16 should be replaced by a statement that inserts as many commas as needed, in the appropriate places.

So how should we rewrite line 12.16?

It might seem that we could add the `g` “global” pattern-matching suffix to the old 12.16, to make the substitution structure say “insert a comma before every triple of digits”; but that will not succeed in this case. Global substitutions work left to right, making a substitution at the leftmost place where they find the pattern and then looking for the next occurrence further rightwards. Inserting commas in numbers has to be done right to left. We don’t put a comma after the first three digits, then after the next three, etc.; we put a comma before the last three digits, then before the three digits preceding that comma, etc. (Numbers are not written like 123,4 or 123,456,78 but like 1,234 or 12,345,678.)

We can handle this with a `while` loop:

```
16  while($population =~ s/(\d)(\d{3})($|,)/$1,$2$3/) {;}
```

The pattern looks for four digits before either the end of the target string or a comma, and inserts a new comma after the first of the four digits. An `=~` construction returns "true" if the pattern is found and "false" if not; so the new line 12.16 will continue to insert commas into a number until the pattern no longer applies anywhere in it – in practice this will mean that the commas are inserted right to left. All the work of the `while` loop is done by the `(condition)` section, so the curly brackets following that section contain just a bare semicolon.

The point here is that, in programming, it usually pays in the long run to do things the right way, even if a "quick and dirty" alternative seems adequate for the moment. In this particular example, admittedly, it might not be difficult to cure lines 12.15–16 if and when we first start working with numbers in the billions. But that is because, inevitably in a short textbook, program (12) is only a simple "toy" example. A program to execute a real-life task will often be much longer; not only will it be considerably more difficult to track down what is going wrong when we adapt it to a new task and find that it fails to perform as expected, but when we do find the problem and try to cure it, the cure will often prove to have its own adverse knock-on effects on other parts of the program, and curing those will create further problems, until it becomes simpler to throw the old program away and start again from scratch. Taking a little extra time to "do things right" from the beginning is much the best policy.