18 Formatted printing

Up to now, our print statements have been simple instructions to print a single string, the only complication that we have seen so far being the possibility of including variable names within the string which are replaced by their values when the print statement is reached:

```
$total = 14726;
print "The total is: $total\n";
```

```
The total is: 14726
```

Often, though, we need more control than this over print formatting, so that complex data sets can be laid out in a way which looks clear to human readers.

This is achieved using the keyword `printf` ("print formatted") rather than `print`. A `printf` statement takes a list of arguments; the first element of the list is a string to be printed (we'll call this the *print string*), subsequent elements identify items to be incorporated into the print string, and the print string contains instructions specifying how to format those other items for inclusion in itself. A formatting instruction is a sequence beginning with the `%` sign, ending with a letter identifying the type of item to be displayed, and (often) having intermediate symbols which "fine-tune" the display format.[34]

For example, the type-letter for integers (whole numbers) is `d;` and a number between the `%` sign and the type-letter specifies a minimum field width. So, rather than writing `print "The total is: $total\n"` above, we could instead have written:

```
printf("The total is:%6d\n", $total);
```

Why might one prefer to do that? Well, for instance, suppose that this line occurs within a loop (so that successive totals will be written out), and suppose that the value of `$total` varies considerably from pass to pass through the loop; then the lines printed out by our `printf` statement will look like this:

```
The total is: 14726
The total is:     3
The total is:   279
```

The symbol `%6d` only says that the *minimum* space to be occupied by the value of `$total` is six characters, so if `$total` should ever get into the millions then the numbers will no longer be neatly aligned with units, tens, etc. one below another. (Normally one would avoid this problem by picking a minimum field width that provides for more places than one ever expects to see.) But, with the `print` statement we showed earlier, the numbers will *never* line up; the display would look like this:

```
The total is: 14726
The total is: 3
The total is: 279
```

In some circumstances, that might be acceptable; but, in others, it could be a thorough nuisance.

Apart from `d`, the "type-letters" most commonly useful are `f` for floating-point numbers (decimals, in ordinary English); `e` for floating-point numbers expressed in scientific notation (e.g. 0.000532 in scientific notation is `5.32e-04`, meaning 5.32×10^{-4}), and `s` for strings.

The most useful intervening symbols, apart from a number standing for minimum field width, are:

`.` followed by a number, for "precision"
`0` use zeros rather than spaces to the left of the number to pad it out to the minimum field width
`-` left-justify rather than right-justify the number within the field

In the case of a floating-point number, "precision" refers to the number of decimal places shown. Thus:

```
$pi = 3.14159265358979;
printf("%07.3f\n", $pi);
```

```
003.142
```

The format symbol `%07.3f` means "print the value to three decimal places and taking up seven character spaces altogether, padding with zeros at the left to achieve that".

Notice that when specifying a limited number of decimal places, we do not need to worry about rounding: Perl does that for us automatically. So in this case 5 in the fourth position after the decimal point correctly causes the preceding 1 to be rounded up to 2.

In the case of strings, "precision" refers to the *maximum* length to be printed:

```
$surname1 = "Smith";
$surname2 = "Cumberbatch";
printf("%.10s\n%.10s\n", $surname1, $surname2);
```

```
Smith
Cumberbatc
```

Perl defines many further "type-letters" and several other intervening symbols, but those are for more specialized purposes.

The items following the print string in a `printf` statement will not necessarily be things that already have names in the program. They may be (and in practice often will be) values that are calculated for the purpose of the `printf` statement. (The same is true for the simple `print` function. Earlier, to keep things simple, we never carried out a calculation within a `print` statement, but that is a quite normal thing to do.)

Consider, for instance, our expanded table of county data, Figure 2 above, which via code-chunk (31) we have read into our program as an array of hashes, so that for instance `$counties[1]{acreage}` gives the value `463830`. Perhaps we would like to know the (average) population densities, i.e. people per acre, of the various counties. We could extract those figures like this:

(32)

```
1   for ($j = 0; $j < @counties; ++$j)
2     {
3     printf("Pop. density of %s is %.3f people per acre\n",
        $counties[$j]{name},
        $counties[$j]{pop}/$counties[$j]{acreage});
4     }

    Pop. density of Bedfordshire is 1.403 people per acre
    Pop. density of Berkshire is 1.262 people per acre
    Pop. density of Buckinghamshire is 1.135 people per acre

      ⋮
```

The print string in the `printf` statement, 32.3, contains two formatting instructions, `%s` and `%.3f` – the latter asks for a floating-point value to be printed to three places of decimals. The expression which provides a string value for `%s` is a simple hash element, `$counties[$county]{name}`; but the following expression, which provides a value for `%.3f`, is a division of one hash element by another hash element.

(As a reminder: in 32.1 it is fine to use the array-name `@counties` in a scalar context, i.e. following `<`, to specify the number of rows in `@counties`: that array really does have one row per county. But what we cannot do is replace the `for($j = 0 ... )` construction of 32.1 with a `foreach` construction which looks at each hash in `@counties` in turn, because what occur in the rows of `@counties` are not really hashes but "references to hashes", and we have not learned how to work with references.)

A final point about `printf` is that printing to the screen is only its common, default use. As with `print`, we can also use `printf` to add material to a file. Thus, if `OUTFILE` is the filehandle for some file that we have opened for appending (`>>`), then

```
printf(OUTFILE "%.10s\n", "Cumberbatch");
```

will add the line

```
Cumberbatc
```

to the end of that file. You might ask "How does `printf` know that in this case `OUTFILE` is the destination file and the print string is the item following that, while in other cases the element immediately following `printf` was the print string? Does this depend on `OUTFILE` not being the name of a string?" No, that is not it; the answer is that the filehandle and the print string are not separated by a comma. If the first item after `printf` has a comma following it, it is the string to be printed and the print destination is `STDOUT`, the "standard output destination" – in practice, the screen. If there is no comma, then that item is the destination file. (In the `printf` statement above, if we added a comma after `OUTFILE` we would get an error message, since `OUTFILE` is in fact a filehandle rather than a string. The important point is that it is presence versus absence of comma which determines how Perl tries to interpret the first item within the brackets.)

To pull everything together, here is a complete program that reads in the countyDataPlus information, stores it in an array of hashes, uses it to calculate the population densities, and saves county names and population densities to an external file. I have included some comments, to make it easier for us to pick up the threads when we come back to the program some time after it was first written. (Adding comments to one's code feels like a chore to most programmers – but trying to recall how uncommented code works usually turns out to be a considerably greater chore!)

(33)

```
1  open(INFILE, "<../data/pops/countyDataPlus.txt") or
   die ("can\'t open countyDataPlus.txt");
2  $i = 0;   #initialize index for rows of @counties
3  while ($line = <INFILE>)
4    {
5    chomp($line);
6    $line =~ s/^\s*(\S.*\S)\s*$/$1/;
7    #remove leading/trailing whitespace
8    @items = split(/\s*\|\s*/, $line);
9    #split on "|" possibly with whitespace adjacent
10   ($name, $population, $acreage, $rateable_value) = @items;
11   $name = despace($name);
12   $counties[$i]{name} = $name;
13   $counties[$i]{pop} = $population;
14   $counties[$i]{acreage} = $acreage;
15   $counties[$i]{rValue} = $rateable_value;
16   ++$i;
17   }
18 close(INFILE);
19 open(OUTFILE, ">../data/pops/densities.txt") or
   die ("can\'t open output file");
20 for ($j = 0; $j < @counties; ++$j)
21   {
22   printf(OUTFILE "%s\t%.3f\n",
       $counties[$j]{name},
       $counties[$j]{pop}/$counties[$j]{acreage});
23   }
24 close(OUTFILE);

25 sub despace
26 #replaces name-internal whitespace with single "_"
27   {
28   my $p = shift(@_);
29   $p = join("_", split(/\s+/, $p));
30   return($p);
31   }
```

If (33) is saved under the name `countyCalcs.pl`, then the command

```
perl -w countyCalcs.pl
```

will create a file `densities.txt` which will look like this:

```
Bedfordshire 1.403
Berkshire    1.262
   ⋮
```

In real life, if all we wanted to do was to discover the population densities, (33) is probably not the program we would write to do that. Setting up an array of hashes containing various kinds of information for each county is a cumbersome procedure if all we are ever going to do is use part of the information for a single calculation, ignoring some of the data (the rateable values) altogether. It would be quicker to do the population-density calculation directly as the individual lines are read in from the countyDataPlus file, and never bother about setting up an array of hashes. Realistically, (33) is more plausible as an early stage of a program which will later be enlarged to process the county data in other ways, perhaps using information from additional input files.

But (33) illustrates, in a small way, everything that real-life programs achieve. It reads data in, processes them, creates data structures to hold them, calculates with them, and saves the results of the processing and calculation to permanent storage. Writing code to achieve these things is what computer programming is about.

19 Built-in variables

In chapter 16, we saw that the symbol `@_` is a special "built-in variable": whenever our program calls a user-defined function, say the function `despace()` which we defined in that chapter, `@_` stands for the list of arguments it is called with. Our program might say:

```
$stringA = "a string of letters";
print(despace($stringA), "\n");
```

```
a_string_of_letters
```

– in this case, while the code headed `sub despace` is being interpreted by the machine, the symbol `@_` (which is used within that code) stands for the one-element array whose single element is `$stringA`.

Perl has numerous built-in variables – a few others are also arrays, most are scalars. Let's look at some of the most useful of them.

Apart from `@_`, the most important built-in array variable is `@ARGV`, which does a job similar to `@_` at the level of the *command line* – the line addressed to your system prompt which tells it to run a Perl program. Our very first program, (1), was a program to add two and two and print out their sum. For a first program that was fine, but in real life it would obviously be more satisfying to have a program which added and printed the sum of *any* pair of numbers we choose to give it. Here is a program to do that; let's name it `printsum.pl`:

(34)

```
1  $a = $ARGV[0];
2  $b = $ARGV[1];
3  print $a + $b, "\n";
```

If we have created a file `printsum.pl` containing (34), we can use it by placing the arguments (that is, for this program, the numbers to be summed) after the program name (without commas):

```
perl -w printsum.pl 2.3 7.9
```

```
10.2
```

The array of arguments to `printsum.pl` is called `@ARGV`, so on this occasion `$ARGV[0]` is 2.3 and `$ARGV[1]` is 7.9.[35]

Better still, we can generalize the program by accepting *any number* of values to be summed – let's call the revised program `printsumm.pl` ("m" for "many"):

(35)

```
1  $t = 0;
2  foreach $item (@ARGV)
3    {
4    $t += $item;
5    }
6  print "$t\n";
```

With `printsumm.pl` defined, we can write:

```
perl -w printsumm.pl 5 19 520 4

548
```

Turning to built-in scalar variables, in fact we have already seen some of these, in chapter 10 on pattern matching. `$1`, `$2`, `$3`, and so on stand for elements identified by round brackets in the pattern section of a pattern-match:

```
$word = "beautiful";
$word =~ /[^aeiou]([aeiou]+)[^aeiou]+([aeiou]+)[^aeiou]/;
# finds the 1st 2 vowel-sequences surrounded by non-vowels
print $1, "\n", $2, "\n";

eau
i
```

Related to these are the built-in variables `$&`, `$`` , and `$'` which, following a pattern-matching operation on a target string, stand for:

`$&`	the section of the target string which matched the pattern
`$``	the preceding section of the target string
`$'`	the following section of the target string

Thus:

```
$word = "beautiful";
$word =~ /eau(..)/;
print $1, "\n";
print $&, "\n";
print $`, "\n";
print $', "\n";

ti
eauti
b
ful
```

The pattern between slashes covers the five characters `eauti` of `$word` (remember that `.` in a pattern stands for any single character); so `$&` stands for that five-character substring. The brackets round `..` mean that `$1` has the value `ti`; `$`` and `$'` stand for the portions of `$word` before and after the segment `eauti`.

Other built-in scalar variables have nothing to do with pattern matching. For instance, `$^T` gives an integer representing the time at which the current program began running (expressed in seconds since the beginning of the year 1970). This huge value may not sound much use in its own right, but for instance we can discover how long a system takes to execute some task by comparing `$^T` with the value returned by `time()`, which is a built-in function giving a count of seconds-since-1970 at the moment when the function call is reached in a program. How long does it take Perl to count to a hundred million? On my machine, six seconds, as measured by the following program `showtime.pl`:

(36)

```
1   for ($i = 0; $i < 100000000; ++$i)
2     {;}
3   print $^T, "\n", time() - $^T, "\n";
```

```
perl -w showtime.pl
```

```
1277200878
6
```

Many built-in scalar variables represent fairly arcane systems-programming concepts, which at this introductory level we can afford to ignore. The most frequently-used built-in scalar variable of all, `$_`, will be passed over briefly here for a different reason. We encountered `$_` once, in chapter 12, in connexion with the `map()` function (where it is indispensable). But the commonest use of `$_` is to provide idiomatically brief alternatives to Perl constructions that take slightly longer to spell out explicitly. For seasoned programmers to whom brevity is important, this may be handy, but beginners are better advised to make their code fully explicit, and hence they should probably avoid using `$_`. (Actually, even professional programmers – not to speak of those who have to maintain their code after they have moved on – are probably better off in the long run making everything explicit at the cost of a few extra keystrokes. There is a geeky side to Perl which delights in terse obscurity for its own sake, and the symbol `$_` is arguably a symptom of that.)

After I have said that much, the reader will doubtless want me to say *something* specific about this use of `$_`, so I will give one example. We know that `foreach` is used to access each element of an array in turn:

(37)

```
1  @colours = ("blue","green","red","yellow");
2  foreach $colour (@colours)
3    {
4    $capColour = uc($colour);
5    print "$capColour\n";
6    }
```

```
BLUE
GREEN
RED
YELLOW
```

Alternatively, it is permissible to omit `$colour` after `foreach`, in which case `$_` is understood:

```
foreach (@colours)
  {
  $capColour = uc($_);
  print "$capColour\n";
  }
```

gives the same output as (37). The symbol `$_` here is like the word *it* in English: the first version of the `foreach` loop was saying something like “for each colour word in the array, change that colour word to upper case”, the second version abbreviated that to something more like “for whatever is in the array, change *it* to upper case”. In English, our speech would quickly become tedious if we spelled everything out rather than using the ambiguous word *it*. But then, in English we negotiate our meaning with one another constantly as we converse, so that ambiguities are eliminated as fast as they arise. In communicating with computers, real ambiguity and real misunderstandings are all too common and hard to avoid. Consequently I would recommend that beginners leave `$_` alone for a while.