6 Program layout

Not everyone sets out the curly brackets on separate lines, as I did in (4) above. Within reason, Perl does not care where in a program we put whitespace (spaces, tabs, and newline characters). Obviously we cannot put a space in the middle of a number – 56237 cannot be written `56 237`, or Perl would have no way to tell that it was all one number [8] – and likewise putting a space in the middle of a string within quotation marks turns it into a different string. But we can set the program out on the page however we please: *around* the basic elements such as numbers, strings, variable names, and brackets of different types, Perl will ignore extra whitespace. Perl will even supply implied spacing in many cases where elements are run together – thus `++ $a` can alternatively be written `++$a`.

Because Perl does not enforce layout conventions (as some languages do), you need to choose some system and use it consistently – so that you can grasp the overall structure of your program listings at a glance.

The main question is about how to indent blocks; different people use different conventions. First, you need to decide how much space you are going to use for one level of indentation (common choices are one tab, or two spaces). But then, where exactly should the indents go? Perl manuals often put the opening curly bracket on the line which introduces it, indent the contents of the block, and then place the closing curly bracket level with the beginning of that first line:

```
if (condition) {
  statement;
  statement;
  ⋮
}
```

This takes fewer lines than other conventions, but it is not particularly easy to read, and it is perhaps illogical in placing the pair of brackets at unrelated positions. Alternatively, one can give both curly brackets lines of their own – in which case they either both line up under the start of the introducing line, or are both indented to align with their contents:

```
if (condition)
{
  statement;
  statement;
  ⋮
}
```

or else:

```
if (condition)
  {
  statement;
  statement;
  ⋮
  }
```

Whichever convention you choose, if you apply it consistently you can catch and correct programming errors as you type. You may have a block which is indented within a block that is itself indented within a top-level block. When you type what you thought was the final }, if it doesn't align properly with the item which it ought to line up with in the first line, then something has gone wrong – perhaps one of your opening brackets has not been given a closing partner?

As for *which* of the three styles you choose, that is entirely up to you. According to Thomas Plum, a survey of programmers working with the similar language C found a slight majority favouring the last of the three conventions.[9] That is the style used in this book.

Indenting consistently also has an advantage when, inevitably, one's program as first written turns out not to run correctly. A common debugging technique is to insert instructions to print out the values of particular variables at key points, so that one can check whether their values are as expected. Once the bugs are found and eliminated, we naturally want to eliminate these diagnostic lines too – we don't want our program spewing out a lot of irrelevancies when it is running correctly. My practice is to write diagnostic lines unindented, so that they stand out visually in the middle of an indented block, making them easy to locate and delete.

The reason to adopt a consistent style for program layout is to make it easier for a human programmer to understand what is going on within a sea of program code – the computer itself does not care about the layout. Another aid to human understanding is *comments*: explanatory notes written by the programmer to himself (or to those who come after him and have to maintain his code) which the machine ignores. In Perl, comments begin with the hash character. A comment can be:

```
# on one or more lines of its own,
# like this
```

or it can be added to a line to the right of code intended for the computer:

```
$total += $a;  # $a is added to the total
```

Either way, everything from the hash symbol to the end of the line is ignored by the machine.