bookboon.com

Perl for Beginners

Geoffrey Sampson

Download free books at

bookboon.com

Geoffrey Sampson

Perl for Beginners

Download free ebooks at bookboon.com

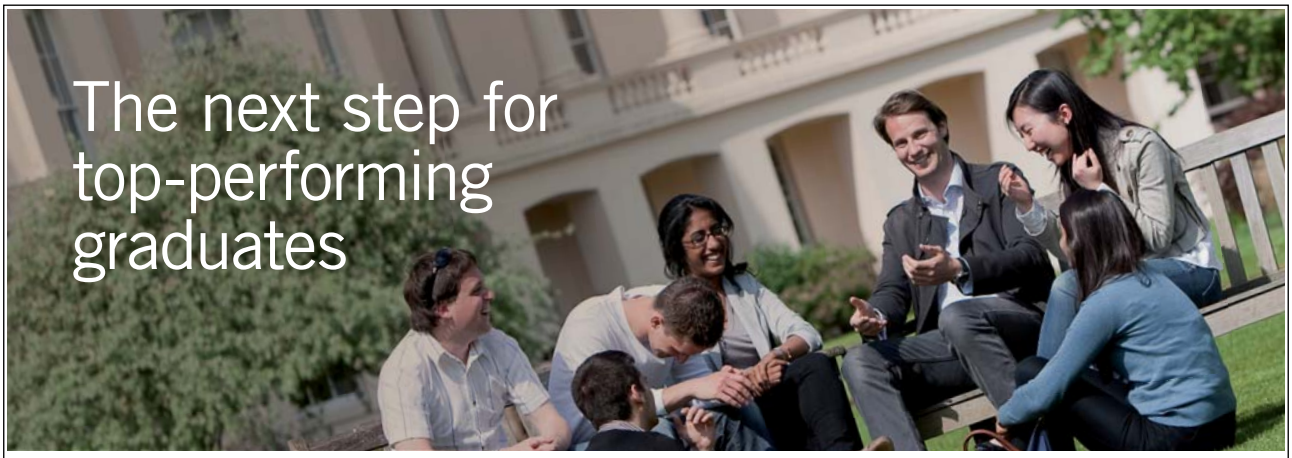
Perl for Beginners
© 2010 Geoffrey Sampson & Ventus Publishing ApS
ISBN 978-87-7681-623-0

Download free ebooks at bookboon.com

Contents

	Note	**8**
1	**Introduction**	**9**
2	**Getting started**	**12**
3	**Data types**	**16**
4	**Operators**	**18**
4.1	Number and string operators	18
4.2	Combining operator and assignment	20
4.3	Truth-value operators	21
5	**Flow of control: branches**	**25**
6	**Program layout**	**26**
7	**Built-in functions**	**28**
8	**Flow of control: loops**	**32**
9	**Reading from a file**	**34**

Please click the advert



Masters in Management

Designed for high-achieving graduates across all disciplines, London Business School's Masters in Management provides specific and tangible foundations for a successful career in business.

This 12-month, full-time programme is a business qualification with impact. In 2010, our MiM employment rate was 95% within 3 months of graduation*; the majority of graduates choosing to work in consulting or financial services.

As well as a renowned qualification from a world-class business school, you also gain access to the School's network of more than 34,000 global alumni – a community that offers support and opportunities throughout your career.

For more information visit www.london.edu/mm, email mim@london.edu or give us a call on +44 (0)20 7000 7573.

* Figures taken from London Business School's Masters in Management 2010 employment report

London Business School

Download free ebooks at bookboon.com

10	**Pattern matching**	**40**
10.1	Matching and substitution	40
10.2	Character classes	41
10.3	Complement classes and indefinite repetition	44
10.4	Capturing subpatterns	46
10.5	Alternatives	47
10.6	Escaping special characters	48
10.7	Greed versus anorexia	48
10.8	Pattern-internal back-reference	50
10.9	Transliteration	51
11	**Writing to a file**	**52**
11.1	Reading, writing, appending	52
11.2	Pattern-matching modifier letters	55
11.3	Generalizing special cases	57
12	**Arrays**	**60**
12.1	Tables with numbered cells	60
12.2.	An example	61
12.3	Assigning a list to an array	64
12.4	Adding elements to and removing them from arrays	64
12.5	Other operations on arrays	66
13	**Lists**	**68**

Please click the advert

You're full of *energy and ideas*. And that's just what we are looking for.

Looking for a career where your ideas could really make a difference? UBS's Graduate Programme and internships are a chance for you to experience for yourself what it's like to be part of a global team that rewards your input and believes in succeeding together.

Wherever you are in your academic career, make your future a part of ours by visiting www.ubs.com/graduates.

www.ubs.com/graduates

UBS

© UBS 2010. All rights reserved.

14	**Scalar versus list context**	**71**
15	**Two-dimensional tables**	**74**
16	**User-defined functions**	**78**
16.1	Adapting Perl to our own tasks	78
16.2	The structure of a user-defined function	79
16.3	A second example	82
16.4	Multi-argument functions	83
16.5	Divide and conquer	84
16.6	Returning a list of values	85
16.7	“Subroutines” and “functions”	88
17	**Hash tables**	**90**
17.1	Tables indexed by strings	90
17.2	Creating a hash	91
17.3	Working through a hash table	94
17.4	Advantages of hash tables	95
17.5	Hashes versus references to hashes	99
18	**Formatted printing**	**101**
19	**Built-in variables**	**107**

Please click the advert

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

Download free ebooks at bookboon.com

20	**The debugger**	**111**
21	**Beyond the introduction**	**115**
	Endnotes	**118**

Please click the advert

Download free ebooks at bookboon.com

Note

All code examples in this textbook have been tested, but it is always possible that bugs may have crept in. Any reader finding an error is warmly invited to let me know, via the e-mail address listed on my website www.grsampson.net – when a revised edition corrects the mistake, you will be acknowledged (if wished) by name.

Geoffrey Sampson
July 2010

Download free ebooks at bookboon.com

1 Introduction

Since its creation in 1987 Perl has become one of the most widely used programming languages. One measure of this is the frequency with which various languages are mentioned in job adverts. The site www.indeed.com monitors trends: in 2010 it shows that the only languages receiving more mentions on job sites are C and its offshoots C++ and C#, Java, and JavaScript.

Perl is a general-purpose programming language, but it has outstanding strengths in processing text files: often one can easily achieve in a line or two of Perl code some text-processing task that might take half a page of C or Java. In consequence, Perl is heavily used for computer-centre system admin, and for Web development – Web pages are HTML text files.

Another factor in the popularity of Perl is simply that many programmers find it fun to work with. Compared with Perl, other leading languages can feel worthy but tedious.

Perl is a language in which it is easy to get started, but – because it offers handy ways to do very many different things – it takes a long time before anyone *finishes* learning Perl (if they do ever finish). One standard reference, Steven Holzner's *Perl Black Book* (second edn, Paraglyph Press, 2001) is about 1300 dense pages long. So, for the beginner, it is important to focus on the core of the language, and avoid being distracted by all the other features which are there, but are not essential in the early stages.

This book helps the reader to do that. It covers everything he or she needs to know in order to write successful Perl programs and grow in confidence with the language, while shielding him or her from confusing inessentials.[1] Later chapters contain pointers towards various topics which have deliberately been omitted here. When the core of the language has been thoroughly mastered, that will be soon enough to begin broadening one's knowledge. Many productive Perl programmers have gaps in their awareness of the full range of language features.

The book is intended for beginners: readers who are new to Perl, and probably new to computer programming. The book takes care to spell out concepts that would be very familiar to anyone who already has experience of programming in some other language. However, there will be readers who use this book to begin learning Perl, but who have worked with another language in the past. For the benefit of that group, I include occasional brief passages drawing attention to features of Perl that could be confusing to someone with a background in another language. Programming neophytes can skim over those passages.

Download free ebooks at bookboon.com

The reader I had in mind as I was writing this book was a reader much like myself: someone who is not particularly interested in the fine points of programming languages for their own sake, but who wants to use a programming language because he has work he wants to get done, and programming is a necessary step towards doing it. As it happens, I am a linguist by training, and much of my own working life is spent studying patterns in the way the English language is used in everyday talk. For this I need to write software to analyse files of transcribed tape-recordings, and Perl is a very suitable language to use for this. Often I am well aware that the program I have written is not the most elegant possible solution to some task at hand, but so long as it works correctly I really don't care. If some geeky type offered to show me how I could eliminate several lines of code, or make my program run twice as fast, by exploiting some little-known feature of the language which would yield a program delivering exactly the same results, I would not be very interested.

Too many computing books are written by geeks who lose sight of the fact that, for the rest of us, computers are tools to get work done rather than ends in themselves. Making programs short is good if it makes them easier to grasp and hence easier to get right; but if brevity is achieved at the cost of obscurity, it is bad. As for speed: computer programs run so fast that, for most of us, speeding them up further would be pointless. (For every second of time my programs take to run, I probably spend a day thinking about the results they produce.)

That does not mean that, in writing this book, I would have been justified in focusing only on those particular elements of Perl which happen to be useful in my own work and ignoring the rest – certainly not. Readers will have their own tasks for which they want to write software, which will often be very different from my tasks and will sometimes make heavy use of aspects of Perl that I rarely exploit. I aim to cover those aspects, as well as the ones which I use frequently. But it does mean that the book is oriented towards Perl programming as a practical tool – rather than as a labyrinth of fascinating intellectual arcana.

If, after working through this book, you decide to make serious use of Perl, sooner or later you will need to consult some larger-scale Perl book – one organized more as a reference manual than a teaching introduction. This short book cannot pretend to cover the reference function, but there is a wide choice of books which do. (And of course there are plenty of online reference sources.) Many Perl users will not need to go all the way to Steven Holzner's 1300-pager quoted above. The manual which I use constantly is a shorter one by the same author, *Perl Core Language Little Black Book* (second edn, Paraglyph Press, 2004) – I find Holzner's approach particularly well suited to my own style of learning, but readers whose learning styles differ might find that other titles suit them better.

Because the present book deliberately limits the aspects of Perl which it covers, it is important that readers should not fall into the trap of thinking "Doesn't Perl have a such-and-such function, then? – that sounds like an awkward gap to have to work round". Whatever such-and-such may be, very likely Perl has got it, but it is one of the things which this book has chosen not to cover.

Download free ebooks at bookboon.com

Having said all that, though, let me stress that what the present book does teach you is not so limited as to be unusable in practice. Far from it. Many, many real-life programming tasks can be very successfully achieved in Perl without venturing beyond the elements of the language covered here. The programming examples you will encounter in this book will all be short programs to carry out little “toy” tasks, to make them easy to learn from; but although programs to achieve real-life tasks will often be *longer* (because the tasks involve more complications), they will not need to be different in kind. This book offers everything you need to begin working as a Perl programmer. Good luck, and have fun!

Download free ebooks at bookboon.com

2 Getting started

For the purposes of this textbook, I shall assume that you have access to a computer system on which Perl is available, and that you know how to log on to the system and get to a point where the system is displaying a prompt and inviting you to enter a command. Perl is free, and versions are available for all the usual operating systems, so if you are working in a multi-user environment such as a university computer centre then Perl is almost sure to be on your system already. (It would take us too far out of our way to go through the details of installing Perl on a home computer which does not already have it; though, if the home computer is a Mac running OS X, it *will* already have Perl – available from the Terminal utility under Applications → Utilities.)

Assuming, then, that you have access to Perl, let us get started by creating and running a very simple program.[2] Adding two and two is perhaps as simple as it gets. This could be a very short Perl program indeed, but I'll offer a slightly longer one which illustrates some basics of the language.

First, create a file with the following contents. Use a text editor to create it, not a word-processing application such as Word – files created via WP apps contain a lot of extra, hidden material apart from the wording typed by the user and displayed on the screen, but we need a file containing just the characters shown below and no others.

```
$a = 2;
$b = $a + $a;
print $b;
```

Save it under some suitable name – `twoandtwo.pl` is as good a name as any. The `.pl` extension is optional – Perl itself does not care about the format of filenames, and it would respond to the program just the same if you called it simply `twoandtwo` – but some operating systems want to see filename extensions in some circumstances, so it is probably sensible to get in the habit of including `.pl` in the names of your Perl programs.

Your `twoandtwo.pl` file will contain just what is shown above. But later in this book, when we look at more extended examples of Perl code I shall give them a label in brackets and number the lines, like this:

(1)

```
1  $a = 2;
2  $b = $a + $a;
3  print $b;
```

These labels will be purely for convenience in discussing the code, for instance I shall write "line 1.3" to identify the line `print $b`. The labels are not part of what you will type to create a program. However, when your programs grow longer you may find it helpful to create them using an editor which shows line-numbers; the error messages generated by the Perl interpreter will use line numbers to identify places where it finds problems.

Download free ebooks at bookboon.com

In (1), the symbols `$a` and `$b` are *variables* – names for pigeonholes containing values (in this case, numbers). Line 1.1 means "assign the value 2 to the variable `$a`". Line 1.2 means "assign the result of adding the value of `$a` to itself to the variable `$b`". Line 1.3 means "display the value of `$b`". Note that each instruction (the usual word is *statement*) ends in a semicolon.

To run the program, enter the command

```
perl twoandtwo.pl
```

to which the system will respond (I'll show system responses in italics) with

```
4
```

Actually, if your system prompt is, say, `%`, what you see will be

```
4%
```

– since nothing in the `twoandtwo.pl` program has told the system to output a newline after displaying the result and before displaying the next prompt. For that matter, nothing in our little program has told the system how much precision to include in displaying the answer; rather than responding with `4`, some systems might respond with `4.00000000000000` (which is a more precise way of saying the same thing). In due course we shall see how to include extra material in a program to deal with issues like these. For now, the point is that the job in hand has been correctly done.

Please click the advert

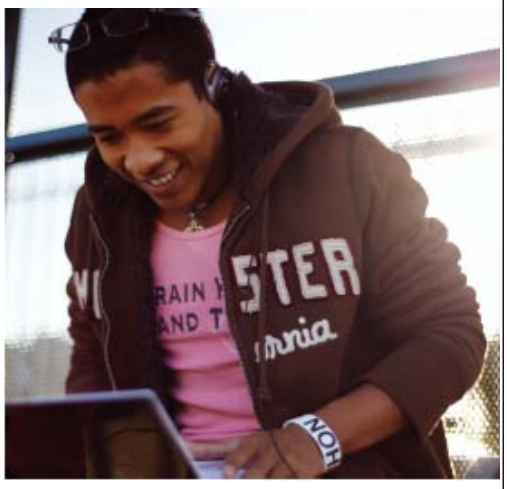
ericsson.com

YOUR CHANCE TO CHANGE THE WORLD

Here at Ericsson we have a deep rooted belief that the innovations we make on a daily basis can have a profound effect on making the world a better place for people, business and society. Join us.

In Germany we are especially looking for graduates as Integration Engineers for
- Radio Access and IP Networks
- IMS and IPTV

We are looking forward to getting your application! To apply and for all current job openings please visit our web page: **www.ericsson.com/careers**



ERICSSON

Download free ebooks at bookboon.com

If you have typed the code exactly as shown and Perl does not respond correctly (or at all) when you try running it, various system-dependent problems may be to blame. I assume that, where you are working, there will be someone responsible for telling you what is needed to run Perl on your local system. But meanwhile, I can offer two suggestions. It may be that your program needs to tell the system where the Perl interpreter is located (this is likely if you are seeing an error message suggesting that the command `perl` is not recognized). In that case it is worth trying the following. Include as the first line of your program this "magic line":[3]

```
#!/usr/bin/perl
```

This will not be the right "magic line" for every system, but for many systems it will be. Secondly, if Perl appears to run without generating error messages, but outputs no result, or outputs material suggesting that it stopped reading your program before the end, it may be that your editor is supplying the wrong newline symbols – so that the sequence of lines looks to the system like one long line. That will often lead to problems; for instance, if the first line of your program is the above "magic line", but Perl sees your whole program as one long line, then nothing will happen when you run it, because the Perl interpreter will only begin to operate on the line following the "magic line". Set your editor to use Unix (decimal 10) newlines.

If neither of these solutions works, then, sorry, you really will need to find that computer-support staff member to tell you how to run Perl on the particular system you are working at!

Let's now go back to the contents of program (1). One point which may have surprised you about our first program is the dollar signs in the variable names `$a` and `$b`. Why not simply name our variables `a` and `b`? In many programming languages, these latter names would be fine, but in Perl they are not. One of the rules of Perl is that any variable name must begin with a special character identifying what kind of entity it is, and for individual variables – names for single separate pigeonholes, as opposed to names for whole sets of pigeonholes – the identifying character is a dollar sign.

If you ask *why* variable names in this particular language should have this strange requirement, the answer has to do with ensuring that the Perl interpreter – the software which "understands" your lines of code and translates them into actions within the workings of the computer – can resolve any line mechanically and without ambiguity. Any programming language has to make compromises between allowing users to write in ways that feel clear and natural to human beings, and imposing constraints so as to make things easy for the computer, which cannot read the programmer's mind and has to operate mechanically. Requiring dollar signs on variable names is a constraint which gives such large clues to the Perl interpreter that it frees the language up to be easygoing and tolerant of humans' preferred usage in other respects. Although many other programming languages have no similar requirements on variable names, overall they are more rigid than Perl about forcing users to code in unnatural ways.

After the dollar sign, a variable name can be any mixture of letters, numbers, and the underline symbol "_", beginning with a letter. (The possibilities are in fact a bit wider than this in some complicated ways, but I am keeping things simple; you will never go wrong by choosing variable names which conform to that pattern.) So e.g. `$fern`, `$fern23`, or `$Fern` would all be good variable names. (Case matters: `$fern` and `$Fern` are two different variables.) In principle there is a limit on the length of a variable name, but you are never likely to bump up against the limit.

Download free ebooks at bookboon.com

The reason for allowing the underline character is so that it can be used to represent a written space when the obvious name for something is a multi-word phrase. If we need a variable to represent, say, roof tiles, we cannot call it `$roof tiles` (which the interpreter would see as a variable `$roof` followed by an unknown word), but we could call it `$roof_tiles`. Alternatively, for the sake of brevity some programmers prefer to run words in variable names together and use capitals to show where they join: `$roofTiles`. It is a good idea to pick one of these two styles which suits you, and to stick to it consistently as your Perl programs grow longer and more complex.

Saving a Perl program in a named file and running it by giving your system prompt the command `perl` *`program-name`*, as we did above, is not the only way to run Perl. If we don't want to take the time to save a short program to a file before testing it, we can simply enter the command `perl` at the system prompt, and then type the program in line by line. In that case, we need to tell the system when we have finished typing; we indicate that by entering `__END__` as the last line, whereupon the system will run the program.

This direct way of running Perl is a good, low-effort method of deepening your mastery of the language by quickly testing brief examples of constructions you are not sure about. To learn Perl, or any other programming language, you have to use the language. No-one ever really *taught* anyone else to program; we all have to teach ourselves, and the most a teacher or a textbook can achieve is to put learners in a position to teach themselves by doing. If you feel unsure how some piece of Perl works, try it, and if it doesn't work the way you expect first time, experiment until it does what you have in mind. That way you will remember it far better than by reading the information in a book.

When you have a program which is thoroughly debugged, so that you are likely to want to run it repeatedly, it is possible to save the effort of typing `perl` on the command line by making the program name itself a recognized command – that is, rather than entering `perl twoandtwo.pl` at the system prompt, you can just enter `twoandtwo.pl`. However, the methods of achieving that vary from system to system, so we shall not look into them here. It does not take much effort to type the word `perl`, after all.

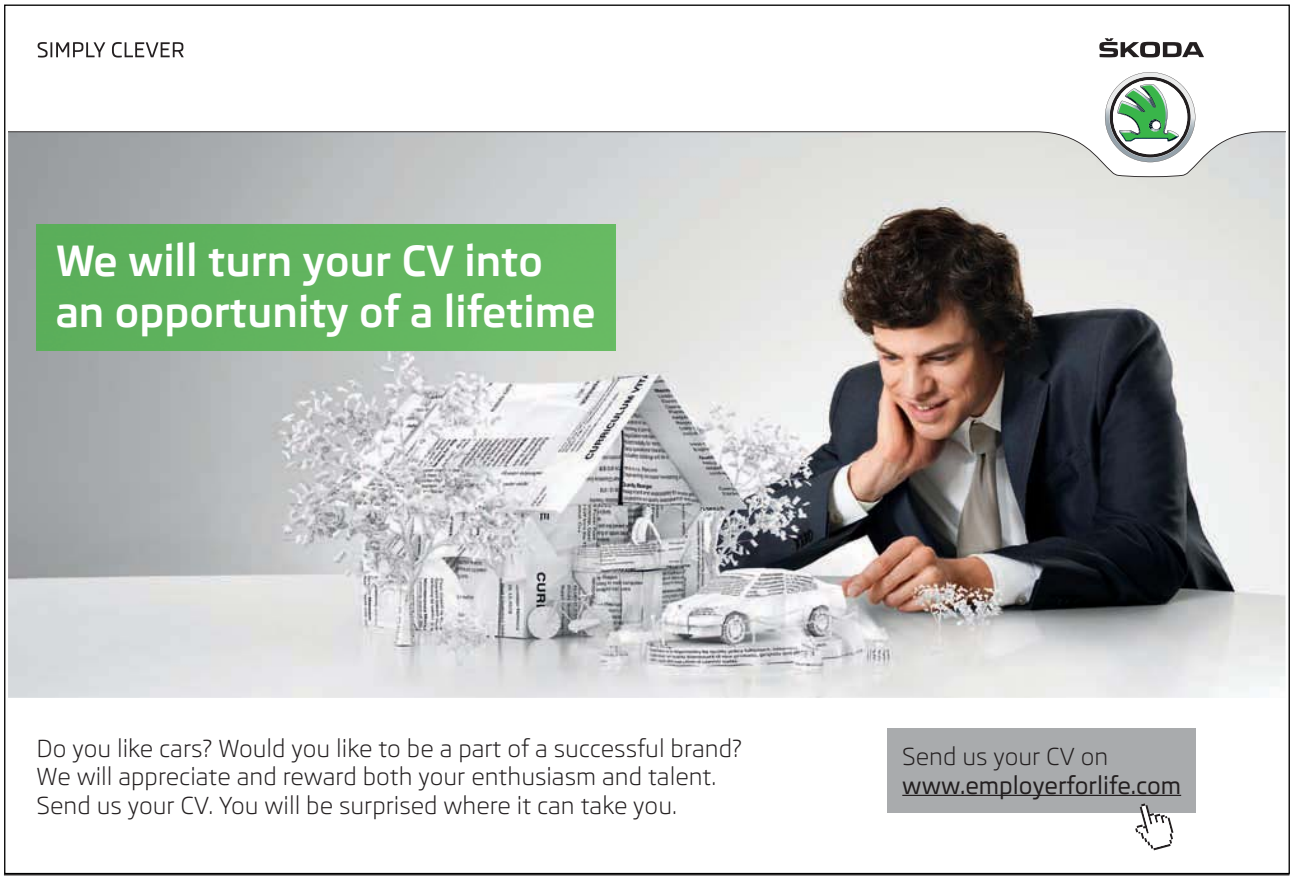
Download free ebooks at bookboon.com

3 Data types

Programming, in any language, involves creating named entities within the machine and manipulating them – using their values to calculate the value for a new entity, changing the values of existing entities, and so forth. Some languages recognize many different kinds of entity, and require the programmer to be very explicit and meticulous about "declaring" what entities he will use and what kind each one will be before anything is actually done with them.[4] In C, for instance, if a variable represents a number, one must say what *kind* of number – whether an integer (a whole number) or a "floating-point number" (what in everyday life we call a decimal), and if the latter then to what degree of precision it is recorded. (Mathematically, a decimal may have any number of digits after the decimal point, but computers have to use approximations which round numbers off after some specific number of digits.)

Perl is very free and easy about these things. It recognizes essentially just three types of entity: individual items, and two kinds of sets of items – *arrays*, and *hashes*. Individual entities are called *scalars* (for mathematical reasons which we can afford to ignore here – just think of "scalar" as Perl-ese for an individual data item); a scalar can have any kind of value – it can be a whole number, a decimal, a single character, a string of characters (for instance, an English word or sentence) ... We have already seen that variable names representing scalars (the only variables we shall be considering for the time being) begin with the `$` symbol; for arrays and hashes, which we shall discuss in chapters 12 and 17, the corresponding symbols are `@` and `%` respectively.

Please click the advert



Download free ebooks at bookboon.com

Furthermore, Perl does not require us to declare entity names before using them. In the mini-program (1), the scalars `$a` and `$b` came into existence when they were assigned values; we gave no prior notice that these variable names were going to be used.

In program (1), the variable `$b` ended up with the value 4. But, if we had added a further line:

```
$b = "pomegranate";
```

then `$b` would have ceased to stand for a number and begun to stand for a character-string – both are scalars, so Perl is perfectly willing to switch between these different kinds of value. That does not mean that it is a good idea to do this in practice; as a programmer you will need to bear in mind what your different variable names are intended to represent, which might be hard to do if some of them switch between numerical and alphabetic values. But the fact that one *can* do this makes the point that Perl does not force us to be finicky about housekeeping details.

Indeed, it is even legal to use a variable's value before we have given it a value. If line 1.2 of (1) were changed to `$b = $a + $c`, then `$b` would be given the sum of 2 plus the previously-unmentioned scalar `$c`. Because `$c` has not been given a value by the programmer, its value will be taken as zero (so `$b` will end up with the value 2). Relying on Perl to initialize our variables in this way is definitely a bad idea – even if we need a particular variable to have the initial value zero, it is much less confusing in the long run to get into the habit of always saying so explicitly. But Perl will not force us to give our variables values before we use them.

Because this free-and-easy programming ethos makes it tempting to fall into bad habits, Perl gives us a way of reminding ourselves to avoid them. We ran program (1) with the command:

```
perl twoandtwo.pl
```

The `perl` command can be modified by various options beginning with hyphens, one of which is `-w` for "give warnings". If we ran the program using the command:

```
perl -w twoandtwo.pl
```

then, when Perl encounters the line `$b = $a + $c` in which `$c` is used without having been assigned a value, it will obey the instruction but will also print out a warning:

```
Use of uninitialized value in addition (+) at twoandtwo.pl line 2.
```

If a skilled programmer gets that warning, it is very likely to be because he thinks he has given `$c` a value but in fact has omitted to do so. And `perl -w` gives other warnings about things in our code which, while legal, might well be symptoms of programming errors. It is a good idea routinely to use `perl -w` to run your programs, and to modify the programs in response to warning messages until the warnings no longer appear – even if the programs seem to be giving the right results.

Download free ebooks at bookboon.com

4 Operators

4.1 Number and string operators

In program (1) we saw the operator `+`, which as you would expect takes a pair of numerical values and gives their sum. Likewise `-` is used as a minus sign. Some further operators (not a complete list, but the ones you are most likely to need) include:

`*`	multiplication
`/`	division
`**`	exponentiation: `2 ** 3` means 2^3, i.e. eight

These operators apply to numerical values, but others apply to character-strings. Notably, the full stop `.` represents *concatenation* (making one string out of two):

```
$p = "witch";
$q = "craft";
$r = $p . $q;
print $r;
```

```
witchcraft
```

(Beware of possible confusion here. Some programming languages make the plus sign do double duty, to represent concatenation of strings as well as addition of numbers, but in Perl the plus sign is used only for numerical values.)

Another string operator is `x` (the letter x), which is used to concatenate a string with itself a given number of times: `"a" x 6` is equivalent to `"aaaaaa"`, `"pom" x 3` is equivalent to `"pompompom"`. (And `"pom" x 0` would yield the *empty string* – the length-zero string containing no characters – which is more straightforwardly specified as `""`.)

Note, by the way, that for Perl a single character is just a string of length one – there is no difference, as there is for instance in C, between `"a"` and `'a'`, these are equivalent ways of representing the length-one string containing just the character `a`. However, single and double quotation marks are not always equivalent. Perl uses backslash as an *escape* character to create codes for string elements which would be awkward to type: for instance, `\n` represents a newline character, and `\t` a tab. Between double quotation marks these sequences are interpreted as codes:

```
print "witch\ncraft";
```

```
witch
craft
```

but between single quotation marks they are taken literally:

```
print 'witch\ncraft';
```

```
witch\ncraft
```

In practice this means that you will almost always want to use double rather than single quotation marks. If you do want to include a backslash character within a string defined within double quotation marks, you code it as `\\`; and likewise `\"` and `\'` code quotation marks that are part of a string. When you display a line you will commonly want to end it with a newline, so that it doesn't run into whatever is displayed next. Thus:

```
print "Don\'t say \"never\".\n";
```

```
Don't say "never".
```

There are *rules of precedence* among the various operator symbols. Thus, the sequence `2 + 3 * 4` will yield the result 14 (not 20), because `*` has higher precedence than `+`. Here the relative precedence probably seems obvious, because it is the same in school algebra: multiplications are done before additions, not the other way round. But it is not always so easy to predict the precedence. Are you confident that you know whether `12 / 3 * 2` would give eight or two? Rather than learning all the precedence rules by heart, it is much easier to avoid the issue by using brackets: `(12 / 3) * 2` is eight, `12 / (3 * 2)` is two.

Please click the advert

With us you can shape the future. Every single day.

For more information go to:
www.eon-career.com

Your energy shapes the future.

e·on

A detailed Perl manual will give the full rules of precedence, together with a number of less-used operators not covered here. But many successful Perl programmers are hazy about a few of the more arcane operators – and I wonder whether *anyone* is confident about every detail of the precedence rules. Brackets are easier.

Incidentally, although the main purpose of an assignment statement, such as `$a = 0`, is to give the symbol on the left a value, Perl regards the entire statement as an expression with a value (its value is the value assigned by the equals sign). This means that if we want to initialize various variables with the same value, we don't need to write separate assignment statements

```
$a = 0;
$b = 0;
$c = 0;
```

– it is enough to write `$a = $b = $c = 0`. An expression like this is interpreted as if it were written `$a = ($b = ($c = 0))`: `$c` is straightforwardly assigned the value zero, then `$b` is assigned the value `($c = 0)`, which is itself zero – and `$a` is assigned the value `($b = 0)`, which is again zero.

4.2 Combining operator and assignment

One thing that a programmer very often needs to do is to change the value of a variable by applying some arithmetic operation to its current value – say, adding the value of another variable:

```
$a = $a + $b;
```

Because this is such a frequent thing, it can be abbreviated by combining the operator and the assignment symbol:

```
$a += $b;
```

and likewise `$a -= $b` means "reduce the value of `$a` by that of `$b`", and so forth:

```
$a = 21;
$b = 3;
$a /= $b;
print $a;
```

```
7
```

Very often, the arithmetic operation consists of either adding or subtracting one; these operations can be further abbreviated to `++` and `--`:

```
$a = 20;
++ $a;
print $a;
```

```
21
```

(There is a subtle difference between `++ $a` and `$a ++`, in terms of when the addition happens. A beginner is recommended always to put `++` or `--` *before* the variable to which it applies, in which case the addition or subtraction is carried out before the variable is used in any further operations.)

4.3 Truth-value operators

The operators seen so far give either a number or a string as their result. There are also operators which yield the answers "true" or "false". To see how these work, consider that very often we want a program to branch: if so-and-so then do *this*, otherwise do *that* (or, do nothing). Branching is handled by a construction like this:

```
if ($a > 100)
  {
  print "It\'s big.\n";
  }
⋮
```

When the program reaches this section of code, it checks whether the current value of `$a` is over 100; if so, the code between curly brackets is executed, i.e. the message is printed out, otherwise that block of code is ignored; and in either case the program then moves on to whatever statements follow after the closing curly bracket. Obviously `>` means "is greater than", so it yields either the value "true" or the value "false": `6 > 5` gives "true", `6 > 6` or `6 > 7` give "false".[5]

The meaning of `>` is straightforward, and likewise `<` means "is less than", `>=` and `<=` mean "is greater/less than or equal to", and `!=` means "is not equal to". The big stumbling block, which often leads experienced programmers into careless mistakes, comes from the fact that, most often, one wants to ask whether some value "is equal to" another. The Perl for "is equal to" is `==` (*two* equals signs).

It is all too easy to write something like:

```
if ($a = 100)
  {
  ⋮
  }
```

thinking that you are testing whether `$a` is equal to 100. You aren't. A single equals sign is the assignment symbol: it means "*make* the thing on my left be equal to the thing on my right". So a computer encountering `if ($a = 100)` will first change whatever value `$a` previously had to the value 100, and then decide what to do with the `if` by considering the "truth value" of 100. A number does not really have a truth-value, of course, but for reasons that we can skip over here Perl will treat the number 100 as "true"; so it will do whatever is given within the curly brackets. Try it:

```
$a = 2;
if ($a = 100)
  {
  print "It\'s one hundred.\n";
  }
```

```
It's one hundred.
```

or even

```
$a = "pomegranate";
if ($a = 100)
  {
  print "It\'s one hundred.\n";
  }
```

```
It's one hundred.
```

I have spelled this out at length, because the mistake is so easy to make. To *test for equality* between numerical values you need *two* equals signs. A single equals sign does not test anything, it *assigns* a value.

A further complication is that `==`, `!=`, `>`, and so forth can only be used to compare *numerical* values. Often, one wants to check whether two *strings* are the same or different. For strings, "is equal to" is symbolized as `eq` (and "is not equal to" is `ne`). Thus:

Please click the advert

Download free ebooks at bookboon.com

(2)

```
1  $a = "pomegranate";
2  if ($a eq "Pomegranate")
3    {
4    print "They\'re the same.\n";
5    }
6  if ($a ne "Pomegranate")
7    {
8    print "They\'re different.\n";
9    }
```

```
They're different.
```

(Lower case versus capital makes the strings unequal.) You must not use `==` or `!=` in 2.2 or 2.6 – if you do, Perl will apply the wrong test (and `perl -w` will issue a warning).

The keywords `eq` and `ne` are the string-comparison counterparts of the number-comparison operators `==` and `!=`.[6] There are also string-comparison counterparts to `<`, `>=`, etc. For instance, when comparing strings, `lt` means "is less than" and `ge` means "is greater than or equal to". What "less" and "greater" refer to in this case is the sorting sequence for strings; for instance, the string `band` is "greater than" `ban` but "less than" `bang`. But this is a specialized kind of string comparison, which many programmers never need to use. For most purposes, `eq` and `ne` are the only string-comparison operators needed.

The symbols `and`, `or`, and `not` apply to expressions which have truth-values to give further truth-values. X `and` Y is true if both X and Y are true, and false if either or both is false. X `or` Y is true if either one of X and Y is true. The expression `not` X is true if X is false, and false if X is true. So, for instance,

```
(3 > 2) and not (4 < 5)
```

gives "false". The expression to the left of `and` is true; but `4 < 5` is true, so the expression to the right of `and`, namely "`not (4 < 5)`", is false. "True `and` false" gives false.[7]

We saw above that Perl includes some shortcuts, such as `*=` or `++`, which achieve conciseness by merging operation and assignment symbolically. There is also one construction which does something similar with a truth-value operator: the three-place `?:` construction. Instead of (2), we could have written:

(3)

```
1  $a = "pomegranate";
2  $b = "They\'re the same.\n";
3  $c = "They\'re different.\n";
4  $d = ($a eq "Pomegranate" ? $b : $c);
5  print $d;
```

```
They're different.
```

What the structure `X ? Y : Z` does is to say "Is X true or false? If it is true, then the value of the whole construction is Y; if X is false, then the value of the whole construction is Z." In this case, the value of `$a eq "Pomegranate"` is "false" (because `$a` begins with lower-case `p`); so `$d` is assigned the value of `$c` rather than that of `$b`, and hence `$c` is what is printed out.

In this toy example, the code using `?:` is not much shorter than the code it replaced. But in realistic programming situations, `?:` can often be very handy.

Please click the advert

Download free ebooks at bookboon.com

5 Flow of control: branches

We have seen the word `if` used to control which instruction is executed next. Commonly, we want to do one thing in one case and another thing in a different case. An `if` can be followed by an `elsif` (or more than one `elsif`), with an `else` at the end to catch any remaining possibilities:

(4)

```
if ($price >= 100)
  {
  print "It\'s expensive.\n";
  }
elsif ($price > 0)
  {
  print "It\'s cheap.\n";
  }
elsif ($price == 0)
  {
  print "It\'s free.\n";
  }
else
  {
  print "Cost is negative.\n";
  print "That can\'t be right!\n";
  }
```

When any one of the tests is passed, the remaining tests are ignored; if `$price` is 200, then since 200 ≥ 100 Perl will print `It's expensive`, and the message in 4.7 will not be printed even though it is also true that 200 > 0.

Curly brackets are used to keep together the *block* of code to be executed if a test is passed. Notice that (unlike in some programming languages) even if the block contains just a single line of code, that line must still have curly brackets round it. The last statement before the `}` does not actually have to end in a semicolon, but it is sensible to include one anyway. We might want to modify our code by adding further statements, in which case it would be easy to overlook the need to add a missing semicolon.

Download free ebooks at bookboon.com

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.

Download free ebooks at bookboon.com

7 Built-in functions

Earlier, we saw that Perl has various "operators" represented by mathematical-type symbols. Sometimes these are the same symbols used in familiar school maths, such as `+` for addition and `-` for subtraction; sometimes they are slightly different symbols adapted to the constraints of computer keyboards, such as `*` for multiplication and `**` for raising to a power; and sometimes the symbols represent operations that we do not usually come across in maths lessons, e.g. "`.`" for concatenation.

Perl has many more built-in functions that could conveniently be represented by special symbols, though.[10] Most are represented by alphabetic codes. For instance, taking the square root of a number is a standard arithmetic operation, but the usual mathematical symbol, √, is nothing like any character in the ASCII character-set, so instead Perl represents it as `sqrt`.

When a function is represented by letters rather than special symbols, the alphabetic code is followed by a pair of round brackets, containing the expression to which the function is applied. Thus:

```
$c = 2;
print sqrt($c);
```

```
1.4142135623731
```

– `$c` here is said to be the "argument" of the function `sqrt()`.

Please click the advert

Download free ebooks at bookboon.com

Perl is fond of offering short-cuts, and commonly it is allowable to omit the brackets round function arguments; `sqrt $c` works as well as `sqrt($c)`. But although Perl does not require the brackets, it is probably a good idea to include them in your programs as a visual reminder of what is going on, at least until you grow in confidence sufficiently not to need that support – and in this book I show a pair of brackets with names of functions, to make it obvious that this is what they are.

(The term `print` is itself really a kind of built-in function, so that to be fully consistent with my own principle I ought to have been writing e.g. `print($a)` rather than `print $a`. Perl happily accepts either; I have made an exception and omitted brackets round arguments to `print`, so as to avoid a confusing piling-up of brackets in a case like `print(sqrt($c))`.)

Other built-in functions stand for operations that have no well-known mathematical symbol. For instance, `int()` gives the whole-number part of a decimal number:

```
print int(3.756);
```

```
3
```

(Notice that `int()` does not round to the *nearest* whole number – it merely throws away the fraction part. We shall see later how to proceed if we want a function that rounds up as well as down.)

I have talked about “operators” and “functions”, but looked at logically these are the same kind of thing. Both of them are devices which accept one or more values as arguments, and use them to deliver a new value. The symbol `*` accepts two arguments and delivers their product. The symbol `sqrt` accepts one argument and delivers its square root. Perl programmers talk about “operators” when the symbol is non-alphabetic and is placed between its arguments, if it has more than one; they talk about “functions” when the symbol is alphabetic and precedes its argument(s). Most “functions” have only one argument, but some have more; most “operators” have multiple arguments, but a few have just one. The distinction is one of terminology only, not a real contrast, and even as a distinction of terminology it is blurry.

We shall not give a comprehensive list of the built-in functions here; this is a topic to explore gradually with the help of a fullscale Perl manual, as your programming needs develop.

Without trying to survey the complete list, it is worth noticing from the start that by no means all functions take a numerical argument and deliver a numerical result, as `sqrt()` and `int()` do.

So, for instance, `length()` gives the number of characters in a string, that is, it takes a string argument and delivers a number:

```
$cabbage = "The quality of mercy is not strained";
print length($cabbage);
```

```
36
```

On the other hand, `chr()` is a function which takes a number as argument and delivers the character whose ASCII code that number is:

```
print chr(65);
```

A

Some functions have strings for both argument and result, e.g. `lc()` makes a string all lower-case and `uc()` makes it all upper-case:

```
print lc("God Save the Queen!");
```

god save the queen!

The function `substr()` takes multiple arguments, normally one string and two numbers, in order to extract a substring from a longer string:

```
$letters = "abcdefghi";
print substr($letters, 4, 2);
```

ef

– the second argument, in this case 4, locates the starting-point within the first argument (counting the initial character of the string as character 0), and the third argument, in this case 2, gives the desired length of substring. Furthermore, while `substr()` most commonly takes three arguments, as here, it can take more – if another string is included as fourth argument, it will replace the substring within the first argument:

```
$letters = "abcdefghi";
print substr($letters, 4, 2, "XYZ");
```

ef

```
print $letters;
```

abcdXYZghi

or it can take fewer – if only the first two arguments are specified, `substr()` will deliver everything from the starting-point onwards:

```
$letters = "abcdefghi";
print substr($letters, 4);
```

efghi

Related to `substr()` is `index()`, which shows where in a string a substring occurs; provided the substring does occur, `index()` returns the location in the longer string of its first character (starting from zero), if it does not occur `index()` gives –1:

```
$a = "curiosity shop";
print index($a, "y s");
print "\t";
print index($a, "ys");
print "\n";
```

```
8   -1
```

A few Perl functions take no arguments at all. The function `rand()` is commonly used with no arguments, to produce a random number between 0 and 1:

```
print rand();
```

```
0.672787631469877
```

– though one can alternatively have the number drawn from the interval between zero and a different upper bound *x* by supplying *x* as an argument to the function:

```
print rand(4.5);
```

```
3.874540028261
```

In all Perl has about eighty built-in functions; a reader who looks through the list in a comprehensive Perl manual will probably find some which are specially useful for his particular application area.

Please click the advert

Download free ebooks at bookboon.com

8 Flow of control: loops

Sometimes we want to repeat an action, perhaps with variations. One way to do this is with the word `for`. Suppose we want to print out a hundred lines containing the messages:

```
Next number is 1
Next number is 2
   ⋮
Next number is 100
```

Here is a code snippet which does that:

```
for ($i = 1; $i <= 100; ++$i)
  {
  print "Next number is $i\n";
  }
```

The brackets following `for` contain: a variable created for the purpose of this `for` loop and given an initial value; a condition for repeating the loop; and an action to be executed after each pass. The variable `$i` begins with the value 1, `++$i` increments it by one on each pass, and the instruction within the curly brackets is executed for each value of `$i` until `$i` reaches 101, when control moves on to whatever follows the closing curly bracket.

We saw earlier that, within double quotation marks, a symbol like `\n` is translated into what it stands for (newline, in this case), rather than being taken literally as the two characters `\` followed by `n`. Similarly, a variable name such as `$i` is translated into its current value; the lines displayed by the code above read e.g. *`Next number is 3`*, not *`Next number is $i`*. If you really wanted the latter, you would need to "escape" the dollar sign:

```
print "Next number is \$i\n";
```

The little examples in earlier chapters often ended with statements such as

```
print $a;
```

In practice, it would usually be far preferable to write

```
print "$a\n";
```

so that the result appears on a line of its own, rather than jammed together with the next system prompt.

Within the output of the above code snippet, 1 is not a "next" number but the first number. So we might want the message on the first line to read differently. By now, we know various ways to achieve that. Here are two – a straightforward, plodding way, and a more concise way:

(5)

```
for ($i = 1; $i <= 100; ++$i)
  {
  if ($i == 1)
    {
    print "First number is $i\n";
    }
  else
    {
    print "Next number is $i\n";
    }
  }
```

or (quicker to type, though less clear when you come back to it weeks later):

(6)

```
for ($i = 1; $i <= 100; ++$i)
  {
  $a = ($i == 1 ? "First" : "Next");
  print "$a number is $i\n";
  }
```

Another way to set up a repeating loop is the `while` construction. Here is another code snippet which achieves the same as the two we have just looked at:

(7)

```
$i = 1;
print "First number is $i\n";
while ($i < 100)
  {
  ++$i;
  print "Next number is $i\n";
  }
```

Here, `$i` is incremented within the loop body, and control falls out of the loop after the pass in which `$i` begins with the value 99. The `while` condition reads `$i < 100`, not `$i <= 100`: within the curly brackets, `$i` is incremented before its value is displayed, so if `<=` had been used in the `while` line, the lines displayed would have reached 101.

The `while` construction is often used for reading input lines in from a text file, so the next chapter will show us how that is done.

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.

Please click the advert

The financial industry needs a strong software platform
That's why we need you

SimCorp is a leading provider of software solutions for the financial industry. We work together to reach a common goal: to help our clients succeed by providing a strong, scalable IT platform that enables growth, while mitigating risk and reducing cost. At SimCorp, we value commitment and enable you to make the most of your ambitions and potential.

Are you among the best qualified in finance, economics, IT or mathematics?

Find your next challenge at
www.simcorp.com/careers

SimCorp

www.simcorp.com
MITIGATE RISK | REDUCE COST | ENABLE GROWTH

Download free ebooks at bookboon.com

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)

```
open(INFILE, "../jjs/weather/annualRecords.txt") or
die("Can\'t open weather data file\n");
while ($line = <INFILE>)
  {
  chomp($line);
  $date = substr($line, 0, 4);
  if ($date < 1970)
    {;}
  elsif ($date > 1979)
    {
    close(INFILE);
    exit;
    }
  else
    {
    print "$line\n";
    }
  }
```

Please click the advert

Download free ebooks at bookboon.com

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:

Download free ebooks at bookboon.com

```
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.

Please click the advert

RedStarRésumé

Do you want your Dream Job?

More customers get their dream job by using RedStarResume than any other resume service.

RedStarResume can help you with your job application and CV.

Go to: Redstarresume.com
Use code "BOOKBOON" and save up to $15

(enter the discount code in the "Discount Code Box")

Download free ebooks at bookboon.com

10 Pattern matching

10.1 Matching and substitution

So far, we have been looking at standard programming functions that just about any language includes. There is nothing very special in the way that Perl implements these. Perl's particular glory lies in *pattern matching*, and we turn to that now.

Pattern matching is about finding particular arrangements of characters within strings (and changing the strings in some way, or taking some other action, when the target patterns are found). Pattern matching in Perl uses the symbol `=~` to link the string being examined (the *target string*) to one of two matching actions, identified by the letters `m` (match) or `s` (substitute):

`m/.../`	return "true" if the pattern ... is present, "false" otherwise
`s/.../.../`	if the pattern ... to the left is found, change it to ... on the right

Pattern matching is about finding the pattern *within* the target string. The pattern usually will not comprise the whole of the target string (though we shall see that, if that is what we want, we can specify that.)

The simplest kind of pattern to look for is a particular substring (though the possibilities become far more sophisticated than that). Let's look at a couple of examples of that simple kind:

```
$a = "location";
if ($a =~ m/cat/)
  {
  print "Found a cat.\n";
  }
```

```
Found a cat.
```

```
$a = "location";
$a =~ s/cat/cut/;
print "$a\n";
```

```
locution
```

I shall call the material represented by dots in `m/.../` and to the left in `s/.../.../` the *pattern* section, and I shall call the material represented by the right-hand dots in `s/.../.../` the *replacement* section.

In the case of the `m/.../` construction, it is permissible to omit the `m`; rather than `if ($a =~ m/cat/)` it works just as well to write `if ($a =~ /cat/)`. This abbreviation saves so little typing that it seems pointless for the language to include it; however, because one can omit the `m`, Perl programmers almost always do omit it – so that readers who move on to other Perl textbooks could be confused if the abbreviation were not mentioned here.[12]

Also with the `m/.../` construction, alongside `=~` for "matches" there is also the symbol `!~` for "does not match":

```
$a = "location";
if ($a !~ /dog/)
  {
  print "No dog here.\n";
  }
```

No dog here.

If the `!~` symbol did not exist, it would of course be easy enough to achieve the same by writing:

```
if (not($a =~ /dog/))
```

but perhaps this abbreviation earns its keep a little better than the omission of `m`.

10.2 Character classes

Matching a specific substring gets us only so far in practice. Things get more interesting when our patterns refer to *classes* of characters, *repetitions* of pattern elements, and so forth.

To make the discussion concrete, let us suppose that (via the INFILE handle) we are reading the text file shown as Figure 1, which is a list of English counties and their populations (in thousands) at the year 1966. (We could of course have used newer data – but it happens that the 1960s data have features which will be helpful for illustrating certain aspects of Perl.)

Intentionally, I have set the file out rather messily – some lines have whitespace before the county name, and the whitespace within some county names and separating names from population figures varies from line to line, containing spaces, tab characters, or both. Text files in real life often are rather messy, and Perl is good for dealing with that kind of mess: messy input will give the pattern matcher good practice.

Download free ebooks at bookboon.com

```
Bedfordshire  428
 Berkshire   585
Buckinghamshire    542
Cambridgeshire and  Isle of Ely    294
Cheshire     1472
Cornwall    353
Cumberland   296
Derbyshire   912
Devon        865
   Dorset    333
Durham  1541
Essex   1244
Gloucestershire    1054
Hampshire    1483
Herefordshire      140
Hertfordshire      872
Huntingdonshire and     Peterborough    184
Kent     1325
Lancashire   5189
Leicestershire 716
Lincolnshire (Holland)  105
Lincolnshire (Kesteven) 226
Lincolnshire (Lindsey) 453
Greater London     7914
Norfolk 586
Northamptonshire   428
Northumberland     828
Nottinghamshire    954
Oxfordshire  349
Rutland 28
Shropshire   322
Somerset     638
Staffordshire1802
East Suffolk 371
West Suffolk 148
Surrey  977
East Sussex  710
West Sussex  450
Warwickshire 2095
Westmorland  67
Isle of Wight      97
Wiltshire    471
Worcestershire     663
Yorkshire, East Riding  543
Yorkshire,   North Riding    584
Yorkshire, West Riding  3736
```

Figure 1

Let's say that we want to extract a list of just those counties with populations of at least a million. Since the numbers shown are thousands, this means in practice: lines containing at least four consecutive digits.

One way to specify a class of characters is by listing them within square brackets. So we could achieve what we want, cumbersomely, by writing:

(9)

```
while ($line = <INFILE>)
  {
  chomp($line);
  if ($line =~ m/[0123456789][0123456789][0123456789][0123456789]/)
    {
    print "$line\n";
    }
  }
```

in which case the machine would respond:

```
Cheshire      1472
Durham  1541
Essex   1244
Gloucestershire    1054
Hampshire     1483
Kent    1325
Lancashire    5189
Greater London     7914
Staffordshire1802
Warwickshire 2095
Yorkshire, West Riding  3736
```

Please click the advert

Try this...

All Options

The sequence 2, 4, 6, 8, 10, 12, 14, 16, ... is the sequence of even whole numbers. The 100th place in this sequence is the number...?

Challenging? Not challenging? Try more ▸▸ www.alloptions.nl/life

Download free ebooks at bookboon.com

But Perl knows the character sequence; a cumbersome expression like `[0123456789]` can be abbreviated as `[0-9]` (and similarly, `[defghij]` could be given as `[d-j]`). So line 9.4 could be reduced to:

```
if ($line =~ /[0-9][0-9][0-9][0-9]/)
```

(remember that `m` for "matches" can be omitted).

Better still, since the class of digits from 0 to 9 is often useful, Perl provides a backslash code `\d` for that class: the line can be shortened further to:

```
if ($line =~ /\d\d\d\d/)
```

Other frequently-useful class codes are `\s` for whitespace characters (space, tab, newline), and `\w` for "word characters", meaning letters of the alphabet, digits, or the underline character (but not e.g. punctuation marks).[13] Capitals in class codes denote the complementary classes: `\D` is any character *other* than a digit, `\S` is any "black" (i.e. non-whitespace) character, `\W` any nonalphanumeric, non-underline character.

Furthermore, rather than repeating `\d` four times, we can specify the number of repetitions wanted within curly brackets:

```
if ($line =~ /\d{4}/)
```

(If we wanted to say "at least 4", so as to allow explicitly for populations of ten million or more, that would be `{4,}`; and `{4,6}` following a pattern element would mean "at least 4 and not more than 6". But this level of explicitness is not needed in the present example; if a string `$line` contains five consecutive digits it must certainly contain four consecutive digits, so it will match the pattern shown above.)

10.3 Complement classes and indefinite repetition

Some counties have multi-word names including spaces. Let's suppose that we want to pick out those lines. We might think of doing that by changing 9.4 so that instead of looking for a sequence of digits, it looks for a whitespace followed by an alphabetic character:

```
if ($line =~ /\s[A-Za-z]/)
```

(Perl offers no backslash code covering letters only; `\w` covers numbers too. `[A-Za-z]` is as concise as we can get for "any (upper or lower case) letter".)

One problem with this, though, is that the names for the divisions of Lincolnshire would be missed: in those names the internal space is followed by an opening bracket rather than a letter. It would be better to define the test as "space followed by any black character other than a digit". We can code this by using the notation `[^...]` meaning "any character *other than* ... ". Our line then becomes:

```
if ($line =~ /\s[^\s\d]/)
```

i.e. the pattern is "whitespace followed by a character that is neither whitespace nor a digit".

However, either of these versions assumes that no line has spaces at the beginning, before the name – any line written that way would get picked by this `if` statement, whether the county name were one word or more – in the present case it would wrongly pick Berkshire and Dorset. As we have seen, it is usually best not to make too many assumptions about where an input file includes whitespace and whether this consists of one or more space(s) and/or tab(s). What we are really looking for is any stretch of whitespace having *non-numeric black characters on both sides*. "One or more" is represented by the plus sign, so the following version of the `if` statement will do the trick:

```
if ($line =~ /[^\s\d]\s+[^\s\d]/)
```

With this line substituted for line 9.4, when we run (9) the output will be:

```
Cambridgeshire and  Isle of Ely    294
Huntingdonshire and     Peterborough    184
Lincolnshire (Holland)  105
Lincolnshire (Kesteven) 226
Lincolnshire (Lindsey) 453
Greater London     7914
East Suffolk 371
West Suffolk 148
East Sussex  710
West Sussex  450
Isle of Wight      97
Yorkshire, East Riding  543
Yorkshire,   North Riding    584
Yorkshire, West Riding  3736
```

Having said that it is safest to allow for messy use of whitespace in input files, perhaps we might actually want to locate lines with initial whitespace, in order to tidy the file up by deleting it. In a pattern, `^` outside square brackets represents the beginning (and `$` represents the end) of a string.[14] So, if `$line` is, say, the string "`  Dorset 333`" (beginning with three space characters), the following substitution statement:

```
$line =~ s/^\s+//;
```

will remove its leading spaces. (`//` as the replacement section changes the substring matching the pattern section to nothing, in other words it deletes that substring.)

If `$line` were a line having no leading whitespace, the statement above would ignore it: `+` means "one or more", so the pattern would not be found and no substitution would occur.

Alongside `+` for "one or more" we have the symbol `*` meaning "zero or more", so we could alternatively write the statement as:

```
$line =~ s/^\s*//;
```

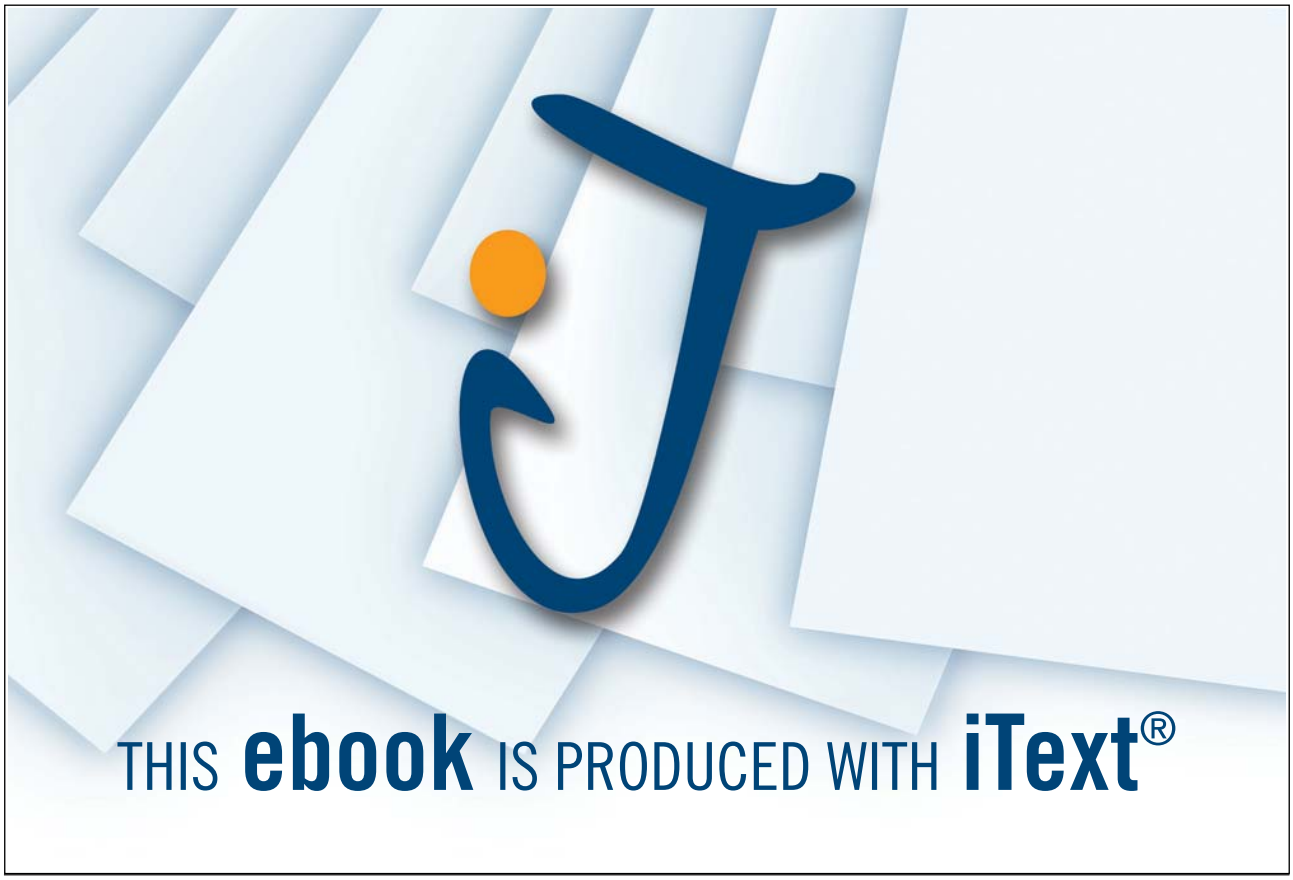
This statement will successfully apply to any line whatever (every line begins with at least zero whitespace characters). But, if `\s*` does match zero whitespace characters, then nothing will be replaced by nothing, in other words there will be no actual change to strings that lack leading whitespace. Here, statements using `+` and `*` are interchangeable in practice; in other situations the distinction between these symbols is crucial.[15]

10.4 Capturing subpatterns

You might object here: if `+` (or `*`) can be any number from one (or zero) upwards, how do we know that `\s+` will match all three leading spaces in a string which contains them, rather than just one or two of them, or none of them if the pattern uses the asterisk? In fact either version of the statement will match all three leading spaces, because Perl pattern-matching is "greedy" – where alternative fits to a pattern are possible, it will always take the one which matches more rather than less of the string under consideration. However, it would clearly be safer to be explicit, and require a black character to follow the whitespace to be eliminated.

But that black character will be the first letter of the county name – we need to retain it in the replacement. Putting round brackets around part of the pattern "captures" whatever substring matches that subpattern so that we can refer back to it in the replacement section: `$1`, `$2`, etc. mean "whatever matched the first/second/... bracketed subpattern".

Please click the advert



Download free ebooks at bookboon.com

Thus, here is a snippet of code which will go through a file removing any leading or trailing whitespace from a line that contains some black characters. It uses the full-stop symbol "`.`", meaning "any character whatever, black or white" – between the first and last black characters of the line there can be any number of intermediate characters, including letters, numbers, and whitespace:

(10)

```
while ($a = <INFILE>)
  {
  chomp($a);
  $a =~ s/^\s*(\S.*\S)\s*$/$1/;
  print "$a\n";
  }
```

The pattern section of the substitution construction in 10.4 specifies beginning-of-line followed by any number of whitespaces followed by a black character, then any sequence of characters of any kind, followed by a black character and then zero or more whitespaces to the end of the line. The replacement section keeps the first and last black characters and whatever came between them. In 10.4 we must write `\s*` rather than `\s+`, because we don't want the statement to apply only to lines that contain *both* leading and trailing whitespace: we want it to apply whenever there is whitespace at either end, or at both ends.

(In our county-population file, we know that the first black character of a line will always be a letter and the last will always be a digit, and we also know that there will always be several characters between the first and last black characters; so (10) would work just as well if the subpattern within brackets were written as `(\w.*\d)` or `(\w.+\d)`. But 10.4 is a more general way of pruning leading and trailing whitespace – it would work equally well for a file in which some lines have punctuation marks at beginning or end, or in which there are only two black characters with nothing between them.)

10.5 Alternatives

The substitution statement 10.4 should work successfully with the material of Figure 1, but it is not as general as it might be. It will ignore a line consisting *only* of whitespace, since there will be no characters for the `\S` symbols to match. Furthermore, it will also ignore a line with just one black character surrounded by whitespace on either side. Each `\S` in the sequence `\S.*\S` must match a separate black character, so the line must contain at least two. In the present context where we are dealing with lists of counties and their populations, although wholly-blank lines might occur, we should be safe in assuming that there will never be a line with just one black character. But suppose that we were dealing with some other kind of input file in which this latter kind of line did occur, and we wanted it too to have leading and trailing whitespace removed. To achieve that, we could modify (10.4) using the symbol "`|`" to represent alternatives:

```
$a =~ s/^\s*(\S.*\S|\S)\s*$/$1/;
```

When we use `|` for "or" in a pattern, normally (as here) we will need round brackets around the alternatives, to show where they begin and end within the pattern. In this case we need the brackets anyway, so that we can use `$1` to refer to the material from first to last black character (or to the sole black character in a line that has only one). But sometimes round brackets will be needed within a pattern just to delimit alternatives – the symbol `$1` (or `$2`, or whatever `$...` symbol corresponds to that bracket-pair) will never be used.

10.6 Escaping special characters

In codes like `\s` (any whitespace character) or `\t` (tab character), the backslash indicates that the following character (`s` or `t`) is not to be given its literal meaning. But where a character such as asterisk or full stop has special pattern-matching significance, a backslash is needed before it when it *should* be given its literal meaning within a pattern. Each of the following characters needs a backslash if it appears with its literal meaning in a pattern:

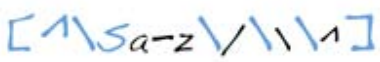
```
\ | ( ) [ { ^ $ * + ? .
```

– and the hyphen needs a backslash if it occurs in its literal meaning between square brackets. However, if you cannot remember precisely which punctuation-type characters require a backslash, it does no harm to put one in anyway – it is only alphanumeric characters which have non-literal meanings when preceded by a backslash in a pattern.

So for instance the class of "all characters other than whitespace, lower-case letters, forward slash, backslash, or circumflex" could be represented as:

```
[^\sa-z\/\\\^]
```

It is logical, but it takes a bit of care to work out precisely what a jumble like that is saying. (When I composed this expression I could not remember whether or not the forward slash needs a backslash in order to be interpreted literally; in fact it does not, but I put one in anyway.) Real-life Perl applications often do tend to need just such complicated patterns involving many non-alphanumeric characters. One tip, when working the pattern out on paper before typing it in, is to write the characters which are used with special pattern-matching meanings larger, and/or in a different colour, than the characters representing themselves:



This helps the eye to grasp the logic of what is going on.

10.7 Greed versus anorexia

We saw that Perl pattern-matching is normally "greedy":

```
$a = "Hertfordshire";
$a =~ /(e.*r)(.*)$/;
print($1, "\n", $2, "\n");
```

```
ertfordshir
e
```

We can make the pattern-matching quantifiers `*`, `+`, etc. "anorexic" by suffixing `?` to them:

```
$a = "Hertfordshire";
$a =~ /(e.*?r)(.*)$/;
print($1, "\n", $2, "\n");
```

```
er
tfordshire
```

This works even with the quantifier `?` (meaning zero or one) itself:

```
$b = "Essex";
$b =~ /E(s?)(.*)x/;
print "$2\n";
```

```
se
```

```
$b = "Essex";
$b =~ /E(s??)(.*)x/;
print "$2\n";
```

```
sse
```

Please click the advert

Masters in Management

Designed for high-achieving graduates across all disciplines, London Business School's Masters in Management provides specific and tangible foundations for a successful career in business.

This 12-month, full-time programme is a business qualification with impact. In 2010, our MiM employment rate was 95% within 3 months of graduation*; the majority of graduates choosing to work in consulting or financial services.

As well as a renowned qualification from a world-class business school, you also gain access to the School's network of more than 34,000 global alumni – a community that offers support and opportunities throughout your career.

For more information visit www.london.edu/mm, email mim@london.edu or give us a call on +44 (0)20 7000 7573.

* Figures taken from London Business School's Masters in Management 2010 employment report

London Business School

Download free ebooks at bookboon.com

10.8 Pattern-internal back-reference

We have seen that the symbols `$1`, `$2`, etc. can be used to refer back to pattern elements demarcated by round brackets, when the `$...` symbols occur outside the pattern – in the replacement part of a `s/.../.../` structure, or in a later code line. Sometimes we need to refer back to a pattern element from a later point *within that pattern*. For that, the symbols are `\1`, `\2`, ... rather than `$1`, `$2`, ...

Here, for instance, is an example of pattern matching which finds, within a string containing names of French towns separated by whitespace, a town name which contains the same pair of adjacent vowels at two separate points in the name:

(11)

```
1   $towns = "Paris Poitiers Bordeaux Toulouse Lyons Marseilles";
2   if ($towns =~ /(\s|^)(\S*([aeiou][aeiou])\S*\3\S*)(\s|$)/)
3     {
4     print "$2\n$3\n";
5     }
```

```
Toulouse
ou
```

The second word in the string, “Poitiers”, contains two pairs of adjacent vowels, but different pairs; the use of `\3` in the pattern-match means that only the fourth word, “Toulouse”, with the same vowel-pair “ou” in two places, fits the pattern.

The pattern in 11.2 takes a bit of unpicking.

First, since the `$towns` string uses whitespace to separate names but does not have whitespace at beginning or end, in order to ensure that every name including the first and the last is tested we need to specify that the substrings examined are preceded either by whitespace or beginning-of-line, i.e. `(\s|^)`, and followed either by whitespace or end-of-line, i.e. `(\s|$)`. These are cases of round brackets which are included in the pattern purely to delimit alternatives: they are respectively the first and fourth pair of round brackets in 11.2, but no use is made of the codes `$1` and `$4`.

Between these two pairs of round brackets is the sequence:

```
(\S*([aeiou][aeiou])\S*\3\S*)
```

This looks for a continuous sequence of black characters, containing a pair of vowels at some point and the same pair of vowels at a later point. Notice that this segment of line 11.2 has one pair of round brackets nested inside another pair. The numbering reflects the sequential order of the *opening* brackets; so the outer brackets surrounding the entire segment are pair 2 (pair 1 was `(\s|^)`, remember), and the internal brackets round `[aeiou][aeiou]` are pair 3. Hence, *within* the pattern, `\3` refers back to whatever `[aeiou][aeiou]` matched (and later, outside the pattern, in line 11.4 `$2` and `$3` refer respectively to the whole word, and to that vowel-pair).

Download free ebooks at bookboon.com

10.9 Transliteration

Finally, apart from the structures `m/.../` and `s/.../.../`, there is a third structure, `tr/.../.../` ("tr" for "transliterate"), which can appear after `=~` and which is commonly discussed in Perl textbooks under the heading "pattern matching". This is misleading, though: what appears between the left-hand pair of slashes in a `tr/.../.../` construction is not a pattern, the whole of which is looked for in the target string, but a sequence of individual characters, any one of which, if found, is replaced by the corresponding character from the right-hand pair of slashes. Thus `tr/.../.../` could be used, for instance, to capitalize each vowel in a string:

```
$a = "triceratops";
$a =~ tr/aeiou/AEIOU/;
print "$a\n";
```

trIcErAtOps

The only real reason to deal with `tr/.../.../` in this chapter rather than somewhere else is that, like `m/.../` and `s/.../.../`, it uses the symbol `=~` as the link to its target string.

Appendix: A checklist of pattern-matching symbols

`[acgt]`	any of the characters `a`, `c`, `g`, `t`		
`[^acgt]`	any character other than `a`, `c`, `g`, `t`		
`[a-t]`	any of the twenty characters between `a` and `t` inclusive		
`.`	any character		
`\d`	digit	`\D`	nondigit
`\s`	whitespace character	`\S`	"black" character
`\w`	"word character"	`\W`	non-word character
`^`	beginning of target string	`$`	end of target string
X`*`	any number (zero or more) *X*'s		
X`+`	one or more *X*'s		
X`?`	zero or one *X*		
X`*?`, *X*`+?`, *X*`??`	as above but matching as little as possible		
X`{3}`	sequence of three *X*'s		
X`{3,}`	sequence of three or more *X*'s		
X`{3,5}`	sequence of at least three and at most five *X*'s		
`(`*X*`|`*Y*`)`	either *X* or *Y*		
`\1`, `\2`, ...	(within a pattern) whatever matched the contents of the 1st/2nd/... pair of round brackets in this pattern		
`$1`, `$2`, ...	(outside a pattern) whatever matched the contents of the 1st/2nd/... pair of round brackets in the last pattern matched		

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).

Please click the advert

You're full of *energy and ideas*. And that's just what we are looking for.

Looking for a career where your ideas could really make a difference? UBS's Graduate Programme and internships are a chance for you to experience for yourself what it's like to be part of a global team that rewards your input and believes in succeeding together.

Wherever you are in your academic career, make your future a part of ours by visiting www.ubs.com/graduates.

www.ubs.com/graduates

UBS

© UBS 2010. All rights reserved.

Download free ebooks at bookboon.com

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.

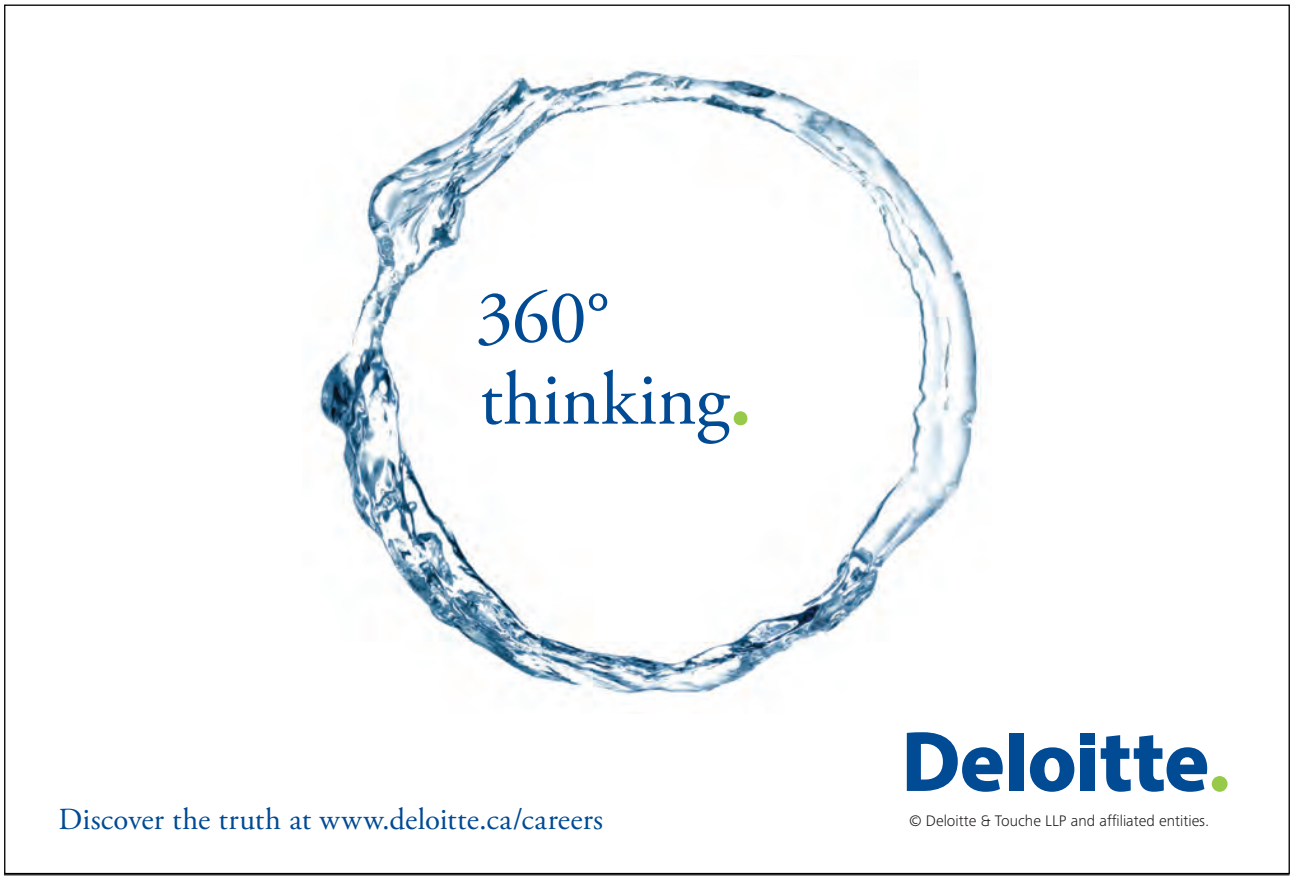
Finally, after the `while` loop has been traversed once for each line of the input file, 12.20–21 tidy things up by explicitly closing the input and output files. (If later code is to read or append to either file, a new `open` statement will be needed.)

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?

Please click the advert



Download free ebooks at bookboon.com

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.

Download free ebooks at bookboon.com

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/) {;}
```

Please click the advert

Download free ebooks at bookboon.com

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.

Download free ebooks at bookboon.com

12 Arrays

12.1 Tables with numbered cells

So far, all our variables have been *scalars*, with names having `$` as their prefix. We said that apart from scalars, for individual data items, Perl has two other types of variable, for organized sets of items: *arrays*, and *hashes*, with prefixes `@` and `%` respectively. The array type is found in most modern programming languages. The hash type is less widespread, more of a Perl speciality (though there are other languages which include it); we shall look at hashes in chapter 17.[17]

An array is like a table, which as a whole has a single name, and within which different pieces of data occupy successive numbered rows. (Tables can have many columns as well as many rows, but for the moment think of an array as a table with a single column of cells.) This being the computing world, the initial cell of the table is numbered zero, rather than one.

Consider again our county-population data. So far, we have seen how to read these data in from a rather messy file, and print them out again as a reformatted, neater file. But we might want to hold the data within our program, so that later activities within the program can refer to various of these data items. One way to do this will be to use the input lines to build up a pair of arrays, `@countyNames` and `@countyPops`, so that `@countyNames` holds the county names in a fixed sequence, and `@countyPops` holds the population figures in the corresponding sequence. Then we can ask a question like "What is the name of county number 11?" by writing:

```
print "$countyNames[11]\n";
```

to which the answer will be:

```
Essex
```

The answer will be `Essex` rather than `Gloucestershire`, because Bedfordshire will be county number 0 not number 1 – see above. More important, notice the dollar sign in the print statement. We have introduced the idea of arrays and said that `@countyNames` is an array, hence its name begins with the `@` symbol. But when we put square brackets after the array name in order to pick out an individual member of the array, the prefix changes to the dollar sign.

This is an odd feature of Perl which beginners usually find confusing, so it deserves a little discussion. Logically, one might expect that if, say, `@fruits` is an array, then the *n*th member of `@fruits` would be identified as `@fruits[n]`, so that one could write a statement like:

```
print @fruits[5];      # wrong!
```

That isn't how Perl works. Because item 5 of the `@fruits` array is an individual item, one has to write:

```
print $fruits[5];
```

damson

– even though the symbol `$fruits` on its own, without following square brackets, refers to nothing.

Apparently, when Perl was created, the inventor believed that users would find this the more natural thing. He was wrong there. It is generally recognized now that this decision was unfortunate (and we are promised that when Perl 6 eventually replaces the current version Perl 5, the decision will be reversed). But we shall be living with Perl 5 for a long time yet, so we just have to get used to this oddity. That is not particularly difficult to do, once we face up to the fact that the rule is different from what most of us instinctively expect.

12.2. An example

Let's now adapt the code (12) which we used to print out the tidied-up county data, so that it instead creates a pair of arrays containing the data: an array `@countyNames` containing the county names, and an array `@countyPops` containing the population figures in the corresponding rows. (This is not the ideal approach – we shall see a better one shortly; but it is the easiest way to start with.)

Please click the advert

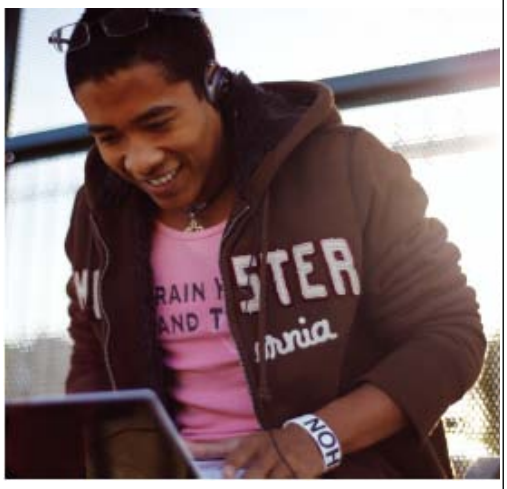
ericsson.com

YOUR CHANCE TO CHANGE THE WORLD

Here at Ericsson we have a deep rooted belief that the innovations we make on a daily basis can have a profound effect on making the world a better place for people, business and society. Join us.

In Germany we are especially looking for graduates as Integration Engineers for
- Radio Access and IP Networks
- IMS and IPTV

We are looking forward to getting your application! To apply and for all current job openings please visit our web page: **www.ericsson.com/careers**



ERICSSON

Download free ebooks at bookboon.com

Where stretches of code do not change, instead of copying them out here I shall just write *[as before]*, so that we can focus on the areas of code which do change.

(13)

```
1  open(INFILE, "<../data/pops/countyData.txt")
      or die "Cannot open input file\n";
2  $i = 0;
3  while ($a = <INFILE>)
4    {
5    if [as before]
6    elsif [as before]
     ⋮
10   else
11     {
12     $name = $1;
13     $population = $2;
14     $name =~ s/(\S)\s+(\S)/$1_$2/g;
15     $population *= 1000;
16     $countyNames[$i] = $name;
17     $countyPops[$i] = $population;
18     ++$i;
19     }
20   }
21 close(INFILE);
```

In line 13.1 we open the input file the same as before, but this time we shall not be printing out a reformatted file, so there is no `open(OUTFILE, ...)`. A new thing we do need to do before embarking on the `while` loop is to set up an index variable, `$i`, which will keep count of positions within the array; on each pass through the loop it will ensure that the name and population of the current county are placed in corresponding rows in the two arrays, and at the end of the loop `$i` will be increased by 1 ready for the next pass.

The initial row of any array is row zero, so 13. 2 sets `$i` to that value. Of course, if human users of the software are going to find it really awkward to think about "county number 0", nothing prevents us giving `$i` the initial value 1, so that Bedfordshire will become county no. 1, and row 0 in both arrays will be "wasted" (it is not like wasting real physical resources!) But if you choose to do that, you will have to remember that you have done it, and make sure that in the future you never carelessly try to use the value of `$countyNames[0]` or `$countyPops[0]` – which would be zero or rubbish values. It is probably easier in the long run to get used to the idea that the initial item of any array is item zero.[18]

Within the `while` loop, there is no change to the way that `if` and `elsif` deal with blank lines and lines with ill-formed contents, or to how the temporary variables `$1` and `$2` are replaced with `$name` and `$population` in the `else` block. Line 13.14 replaces spaces in county names with underline characters as before. But we do not want the population figures to be given commas grouping them into threes, as in the old line 12.16. That was done to turn a number into a string easily grasped by human eyes; but in our `@countyPops` array we shall want the populations to be actual numbers that the machine can calculate with, so that subsequent program code will be able to use the numerical data. So our new line 13.15 simply multiplies the figures taken from the input file by 1000, so that counts of thousands become counts of individuals, rounded to the nearest thousand.

Then, 13.16 puts the current county name in row `$i` of `@countyNames`, 13.17 puts the current county population in the corresponding row of `@countyPops`, and 13.18 increments `$i`, ready for the next pass through the loop.

Notice that we did not have to "warn" Perl that we were going to set up arrays `@countyNames` and `@countyPops` – as soon as a value is first assigned to some row of an array, if the array does not yet exist it is automatically created without further ado. (In this case we did not even mention the names `@countyNames` and `@countyPops` with `@` prefixes; but the square brackets in e.g. `$countyNames[$i]` tell Perl that what precedes them is an array name.)

Line 13.21 closes INFILE as before, but since we never opened an output file, of course we do not now have to close it.

The chunk of code (13) is not intended as a complete program on its own. That does not mean that it lacks some particular elements which Perl requires to occur in any complete program. It doesn't; a "program" in Perl is simply a sequence of statements, so we could call (13) a program if we wanted to. But as a complete program (13) would be no practical use, because it produces no output. It is intended as the beginning of a program, which uses the countyData file in order to create data structures (the arrays `@countyNames` and `@countyPops`) which can be used by code that will be added later in the program, in order to do whatever we may want to do with the data.

Obviously, we can print out individual values:

```
print($countyNames[36], "\t", $countyPops[36], "\n");
```

```
East_Sussex     710000
```

Often, we shall want to write code to do something to *each* element of an array in turn, and chapter 14 will show us how to do that.

(Since (13) is not intended as a complete program, but it seems too long to call a "snippet", I shall call it a "code-chunk". Later chapters will extend (13) into programs which produce output.)

Download free ebooks at bookboon.com

12.3 Assigning a list to an array

Code-chunk (13) put elements into the `@countyNames` and `@countyPops` arrays one by one; but sometimes we will want to set up an array and initialize its contents in one go. The easiest way to do that is to put on the right of the equals sign a list of the array members, separated by commas and with round brackets around the list:

```
@words = ("the","quality","of","mercy","is","not","strained");
@numbers = (1, 2, 3, 4, 5);
```

Alternatively we could of course put the items in one by one, e.g.:

```
$words[0] = "the";
$words[1] = "quality";
   ⋮
$words[6] = "strained";
```

but the bracket-and-comma notation takes less typing.

We can display the contents of an array by giving an array name as argument to a `print` statement. But, if we do this in the obvious way, what gets displayed are the array elements and nothing else. Thus, if `@words` and `@numbers` have the values as above:

```
print(@words, "\n", @numbers, "\n");
```

```
thequalityofmercyisnotstrained
12345
```

This is perfectly logical, but not very user-friendly. The trick is to include the array names within the double quotation marks, in which case the array elements are displayed separately:

```
print "@words\n@numbers\n";
```

```
the quality of mercy is not strained
1 2 3 4 5
```

12.4 Adding elements to and removing them from arrays

We have seen how to create an array and populate it with a long list of values. But often, we want to work with individual items of an array. We already know how to do that using the square-bracket notation. Continuing with the array `@words` as above, the statement:

```
$a = $words[3];
```

will assign the string `"mercy"` to `$a`, leaving `@words` unchanged. The statement:

```
$words[3] = "generosity";
```

will change element 3 of `@words` from `"mercy"` to `"generosity"`, without altering the number of elements in `@words`.[19]

Quite commonly, though, we want to add an element or elements to an array, increasing its length; or to remove element(s) from an array, using their values and leaving the array shorter.

The functions `push()` and `pop()` respectively add elements to and remove elements from the back of an array. The functions `shift()` and `unshift()` respectively remove elements from and add elements to the front of an array. The "remove" functions `pop()` and `shift()` remove one item at a time from the respective array-end and return it as their value; but the "add" functions `push()` and `unshift()` can be used to add either a single item or a list of items.

This is most easily illustrated by example. In code-snippet (14), the right-hand column shows what the array `@colours` contains after each successive statement is executed:

Please click the advert

SIMPLY CLEVER

ŠKODA

Do you like cars? Would you like to be a part of a successful brand?
We will appreciate and reward both your enthusiasm and talent.
Send us your CV. You will be surprised where it can take you.

Send us your CV on
www.employerforlife.com

Download free ebooks at bookboon.com

(14)

```
1  @colours = ("red", "green");        red green
2  $colour1 = shift(@colours);         green
3  push(@colours, "blue", "grey");     green blue grey
4  $colour2 = pop(@colours);           green blue
5  unshift(@colours, "red", "black"); red black green blue
6  print "$colour1\t$colour2\n";
```

```
red grey
```

(Notice that when a list of items is "unshifted" onto the front of an array, the items end up in the array in the same order as they were in the list. Line 14.5 does not first move `"red"` onto the front of `@colours` and then move `"black"` into the new front position before `"red"`.)

12.5 Other operations on arrays

We can create a shorter array as a slice from the middle of a longer array:

```
@words = ("the","quality","of","mercy","is","not","strained");
@phrase = @words[1..3];
print "@phrase\n";
```

```
quality of mercy
```

The word *the* is element 0 of `@words`, so *quality* is element 1; note the use of two dots (rather than three dots as in ordinary English) to separate the end-points of the slice.

The functions `split()` and `join()` convert between strings and arrays. The former makes a string into an array of strings; it standardly takes two arguments: a pattern, between slashes (as in pattern-matching), and the string to be split – the pattern is used to identify places where the string should be split. Thus:

```
$Rev = "At Flores in the Azores Sir Richard Grenville lay";
@words = split(/ /, $Rev);
print($words[1], "\t", $words[4], "\n");
```

```
Flores  Azores
```

Here, the pattern `/ /` (i.e. a single space character between slashes) means "treat spaces as places to split".[20] If the pattern were `//`, the empty pattern, then the string would be split into its individual characters, including spaces.

Conversely, `join()` links the elements of an array into a single string, with the array elements separated by the first argument:

```
@words = ("the","quality","of","mercy");
$colonString = join(":", @words);
print($colonString, "\t", length($colonString), "\n");
```

```
the:quality:of:mercy    20
```

The functions `split()` and `join()` give us an alternative to line 14 of (13) as a way to replace whitespace in a string by single underline characters. We could change 13.14 to:

```
@nameparts = split(/\s+/, $name);
$name = join("_", @nameparts);
```

– i.e. treat sequences of one or more whitespace characters as places to split the line, then join the resulting array `@nameparts` using the underline character. This two-line sequence can be collapsed into one line:

```
$name = join("_", split(/\s+/, $name));
```

– here the second argument to `join()` is whatever results from applying `split()` to its arguments. Even as a single line, this is no shorter than the original line 13.14 – it is rather longer, in characters; but its logic is easier to grasp.

(I make no apology for the original version of 13.14, though: that offered a useful illustration of the workings of the Perl pattern matcher. Often one *must* use the pattern matcher – no convenient alternative using built-in functions will do the job.)

A particularly powerful built-in function applicable to arrays is `map()`, which produces a new array by applying a specified operation to each element of an existing array. The first argument to `map()` defines the operation to apply, as an expression within which the element to which it is applied is represented by the symbol `$_`. The second argument gives the array whose elements are to be operated on. Thus:

(15)

```
1  @numbers = (1, 2, 3, 4, 5);
2  @words = ("end","of","chapter","twelve");
3  @squares = map($_**2, @numbers);
4  @capWords = map(uc($_), @words);
5  print("@squares\t@capWords\n");
```

```
1 4 9 16 25  END OF CHAPTER TWELVE
```

13 Lists

Chapter 12 was rather glib about *lists* – sequences of items separated by commas and surrounded by round brackets. That notation appeared in the last chapter both with print statements:

```
print($countyNames[12], "\t", $countyPops[12], "\n");
```

and also for initializing the various values of an array:

```
@numbers = (1, 2, 3, 4, 5);
```

In previous chapters, `print` was always followed by a single string or variable name. What exactly does it mean to link a list of items with commas and surround it with round brackets?

Please click the advert

With us you can shape the future. Every single day.

For more information go to:
www.eon-career.com

Your energy shapes the future.

e·on

Lists in Perl are treated by some textbook writers as another type of data structure alongside scalars, arrays, and hashes (which we have not covered yet). For instance, a `print` statement in general takes a list of items to print – where just one item is printed, this is a special case (though a very frequent one). However, unlike scalars, arrays, and hashes, a list is not something that has its own variable name (so there is no prefix symbol for lists, parallel to `$`, `@`, and `%`). We can form a list by naming its elements, separated by commas, between a pair of round brackets, but we cannot refer to it using a single symbol. Really, lists are a variety of array, which contains elements in a fixed sequence, but which is unnamed and unnamable, and which does not continue in existence after the statement which creates it (as an array does).[21]

If we want to turn a list into a named entity with continuing existence, we can assign it as the value of an array; as we have seen, this is the easiest way to populate an array with elements. We can also use a list of variables on the left of an equals-sign in order to assign values to several variables at once:

```
($a, $b, $c) = (15, 20, 25);
```

is a good alternative to:

```
$a = 15;
$b = 20;
$c = 25;
```

Normally when assigning a list of values to a list of variables there would be the same number of items in the two lists. A statement such as:

```
($a, $b, $c) = (15, 20);
```

would destroy any value that `$c` had previously, without giving it a new value. And the statement:

```
($a, $b) = (15, 20, 25);
```

simply makes no use of the value 25.

Strictly, if the list on the left-hand side of an assignment includes an array variable, because an array can contain any number of items there can be more values in the right-hand list than variables in the left-hand list. But we have to be very careful here. One might imagine that

```
($mother, @twins, $child3, $child4)
  = ("Mummy", "Topsy", "Tim", "Tansy", "Teddy");
```

would lead to the following assignments:

```
$mother       "Mummy"
@twins        "Topsy", "Tim"
$child3       "Tansy"
$child4       "Teddy"
```

But it won't; it will actually give these assignments:

```
$mother       "Mummy"
@twins        "Topsy", "Tim", "Tansy", "Teddy"
```

and neither `$child3` nor `$child4` will be assigned any values. The first array variable in the left-hand list will always mop up any values on the right that have not already been used. It is best to avoid this problem, by including only scalar variables in lists on the left-hand side of equals signs. The statements:

```
($mother, $child3, $child4) = ("Mummy", "Tansy", "Teddy");
@twins = ("Topsy", "Tim");
```

achieves the desired set of scalar and array variable assignments in a way that leaves no room for confusion.

If there are array variables in a list on the *right*-hand side of an assignment, another pitfall arises. Lists "flatten out" any multi-element variables within them into their individual elements. Suppose we have given the arrays `@words` and `@numbers` the same values that we gave them in section 12.3:

```
@words = ("the","quality","of","mercy","is","not","strained");
@numbers = (1, 2, 3, 4, 5);
```

Then the statement

```
@wordsAndNums = (@words, @numbers);
```

will not create a two-member array; `@wordsAndNums` is now a twelve-member array, with `$wordsAndNums[0]` being `"the"` and `$wordsAndNums[11]` being 5, because the bracket-and-comma notation to the right of the equals sign has flattened the separate arrays into one long list of scalars. (We shall see in chapter 15 that we can have arrays whose members are themselves arrays, but this is not the way to create them.)

All in all, although in one way it is reasonable to explain lists as temporary, nameless arrays, this really understates the difference between the two concepts. Rather than pursue the issue in more detail, it will be best to leave the topic here and to suggest that readers avoid being too imaginative in how they deploy lists within their programs. Using lists in the ways they are used in our examples should not create problems, but doing other kinds of thing with lists can lead to unexpected results. We can always assign a list of values to a named array and work with that; the behaviour of arrays is easier to predict.

Download free ebooks at bookboon.com

14 Scalar versus list context

Often, we shall need to work through an array, in order to process all the entries in some way or to look for entries having some feature of interest. Let's return to the idea of identifying counties with populations over a million, and let's now make it a task to be done in code that follows on from code-chunk (13). In chapter 10 we used pattern matching to count sequences of digits in lines directly they were read in from the external data file. But (13) set us up with arrays containing the population information, which remain available for consultation as long as our program is running – we don't need to go back to the external file. If we knew how many rows there were in the `@countyPops` array, one way to list the million-plus counties would use a `for` loop:

(16)

```
for ($j = 0; $j < N_rows; ++$j)
  {
  if ($countyPops[$j] >= 1000000)
    {
    print ($countyNames[$j], "\n");
    }
  }
```

```
Cheshire
Durham
Essex
   ⋮
```

Please click the advert

Download free ebooks at bookboon.com

Here I am temporarily writing *N_rows* in italics to stand for whatever number of rows the array contains. The initial row is row 0, so the number of the final row will be one less than *N_rows*; therefore this `for` line will run through all the rows of `@countyPops` and stop after the final row, as we want. So what Perl expression should we write in place of *N_rows*?

In fact (see Figure 1) there will be 46 rows, from row 0 for Bedfordshire to row 45 for the West Riding of Yorkshire. But we don't want to hard-wire the number 46 into our program, otherwise the program will break as soon as we vary the data-set – say, by giving it Scottish rather than English counties. (One wise software-engineering expert urges that the *only* numbers which should ever appear as specific numbers in program code are zero and one.[22] That is possibly an extreme point of view, but certainly it would be foolish to incorporate the number 46 into our code explicitly in a case like this one.) What we want in place of *N_rows* is an expression that means "the number of rows in `@countyPops`, *whatever that number happens to be*".

We can achieve this very simply: the expression `@countyPops` itself is all that is needed. Line 16.1 can read:

```
for ($j = 0; $j < @countyPops; ++$j)
```

How can this be? `$j` always represents an integer (0, 1, 2, and so on); how can an individual integer be "less than" (or more than, or equal to) a whole array of numbers?

Here we come to one of the most distinctive features of Perl: the fact that a given term can have different meanings in different contexts. The position following a `<` symbol is a *scalar context*: if we write `$j < ...`, then whatever expression we put in place of the dots has to provide a scalar value. An array is not a scalar, but an array name in a scalar context will be understood by Perl to mean "the number of elements in the array" – and this of course *is* a scalar value.

The alternative to scalar context is *list context*. For instance, `print ...` is a list context, so whatever we put in place of the dots will be interpreted by Perl as a list. If we write, say:

```
print 100;
```

we seem to be asking Perl to print an individual (scalar) value, but Perl will not interpret the request that way; Perl will understand us as asking it to print a list that happens to be one element long.

In this latter case the difference is perhaps academic. So long as Perl actually prints out `100` as expected, why should we care whether it "thinks" it is printing out a list of one element or an individual item? Going in the other direction, though, it can be tricky to know how Perl will interpret a non-scalar item that it encounters in a scalar context. You just have to learn that, for instance, what an array gives in a scalar context is its size – you could not reliably predict this from more general features of the Perl language.[23]

Download free ebooks at bookboon.com

Consequently, at the elementary level we are concerned with in this textbook, we shall not make much use of the ability to change the usual meanings of expressions by putting them into unusual contexts. But this is one of those features that is used so heavily, by people who are proficient in Perl, that beginners might be confused if we did not even mention it here. And the specific example of putting an array name into scalar context to find its size is so useful that even beginners need to know about it. If we want a scalar variable `$N_counties` to contain the number of counties in the array `@countyPops`, we might have expected that Perl would have a built-in function with a name such as `size()` that would take an array as argument and return the number of elements it contains. There is in fact no function `size()`; the nearest equivalent is the function `scalar()`, which gives whatever value its argument gives in a scalar context – so that, if the argument is an array, `scalar()` gives its size. We can assign `$N_counties` a value by writing:

```
$N_counties = scalar(@countyPops);
```

But for that matter, we can achieve the same result by writing simply:

```
$N_counties = @countyPops;
```

– assignment to a scalar variable is a "scalar context", so `@countyPops` will yield its size.

The point about how Perl interprets an array in a scalar context arose because we wanted to find some way of specifying the number of elements in the `@countyPops` array, in order to allow a `for` loop to work through the array and stop when it reached the end. This made a convenient introduction to the issue of finding the size of an array, which is something we often need to do. But, before moving on, we should note that there is a more direct method of working through the elements of an array, which bypasses the array-size issue because it dispenses with the need for a named index variable such as `$j`. Rather than using `for`, to work through each of the entries in an array we can use a special-purpose keyword `foreach`, which is relevant only to arrays.

Suppose we want to calculate the total national population, by adding together the populations of each of the counties (again, by adding code to the program which began with code-chunk (13)). We can do it like this:

(17)

```
1  $totalPop = 0;
2  foreach $countyPop (@countyPops)
3    {
4    $totalPop += $countyPop;
5    }
6  print "$totalPop\n";
```

```
45373000
```

The `foreach` line looks at each row of `@countyPops` in turn, assigning its value to the variable `$countyPop`; successive values of that variable are added to `$totalPop`, which begins as zero and ends containing the overall cumulative total.

15 Two-dimensional tables

In our work with the county data, I said that setting up two arrays, one for the county names and one for their populations, was not the ideal approach. The trouble is that there is nothing but numerical position in the respective sequences to tie individual population figures to the correct names. This is risky. Code-chunk (13), which set the arrays up, will never put a population figure in a wrong cell, differently-numbered from the cell containing the corresponding name; but, in real-life programming, data sets often have to be changed and updated after they are initially created, and we have set up a system here where it would be all too easy for the `@countyNames` and `@countyPops` arrays to get out of synch with one another. It *shouldn't* happen – but, with software, if something can go wrong then sooner or later it probably will.

We are really dealing with a single set of data: on paper it would be one two-dimensional table, with many rows (one for each county) and two columns. So it ought to be coded as a single array, `@counties`, within which an individual county's name and population will be a single data element. And that is easy to do.

Until now, array elements have always been scalars – numbers, or strings. But in Perl they can be anything we like, including arrays.[24] What we want to do is to set up a `@counties` array, within which each row is a two-element array containing a name and a population figure. Then we have eliminated the risk of parallel arrays getting out of synch with one another. We will have mapped the two-dimensional shape of the table on paper into an electronic equivalent.

Please click the advert

Download free ebooks at bookboon.com

Within an array of arrays, the value in the *j*'th column of the *i*'th row will be identified by two pairs of square brackets:

```
$arrofarr[i][j]
```

So, to create the `@counties` array, we need to replace lines 16 and 17 of (13) with the following:

```
$counties[$i][0] = $name;
$counties[$i][1] = $population;
```

Thus our new version of code-chunk (13) will be:

(18)

```
1   open(INFILE, "<../data/pops/countyData.txt")
      or die "Cannot open input file\n";
2   $i = 0;
3   while ($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 = join("_", split(/\s+/, $name));
15      $population *= 1000;
16      $counties[$i][0] = $name;
17      $counties[$i][1] = $population;
18      ++$i;
19      }
20    }
21  close(INFILE);
```

Now that we have our two-dimensional table encoded as an array of arrays, we can pick out an individual scalar data item from it using two pairs of square brackets:

```
print "$counties[40][0]\n";
```

```
Isle_of_Wight
```

```
print "$counties[40][1]\n";
```

```
97000
```

However, I said in the footnote above that Perl only behaved *as if* it included arrays of arrays. What we have actually done with (18) is not exactly what we seem to have done, and you need to be aware of this so that you don't get puzzled when other things you might think of doing with arrays of arrays fail to work.

The elements in the rows of `@counties` are not actually arrays, i.e. sets of data values; they are single items called *references*. We can think of a Perl reference as a sort of "pointer", telling Perl where to find some other item (in this case, where to find another array).[25] In reality, Perl arrays can be only one-dimensional. But, because array elements can be references to sets of data values, a Perl array can *appear* to be multi-dimensional.

One way in which this difference becomes noticeable in practice is that, if you already have an array, you cannot assign it as a whole to be an element of a larger array. So, for instance, you might have thought that rather than putting county name and county population separately into separate cells of the two-dimensional `@counties` array, we could instead for each county have created a two-element array, and then assigned that array as a row of `@counties`. In other words, we might have written the following instead of 18.16–17:

```
16  @county = ($name, $population);
17  @counties[$i] = @county;    # NO!
```

That won't work. Line 16 will create a two-element array `@county` containing a string and a number; but line 17 will not make that array the value of an element of `@counties`. If you are running Perl with warnings (using `perl -w`) you will be warned that line 17 ought to begin with `$`, not `@`, because any element of an array must be a scalar; and if you try printing out, say, `$counties[4][0]`, no county name will emerge.

Provided you stick to building up an array of arrays individual cell by individual cell, though (as we did in (18), where line 16 put a string into column 0 and line 17 put a number into column 1 for each county), Perl is organized so that you do not need to worry about the machinery which creates the appearance of an array whose rows are themselves multi-element arrays. And in this book we shall not worry about it. References are probably the most important Perl topic which this book deliberately avoids discussing in detail – the reason being that beginners usually find it a very confusing topic, and we can achieve quite a lot in Perl without learning about references.

If you progress with Perl, sooner or later you will need to learn about references and "dereferencing". These topics are needed for other purposes as well as multi-dimensional arrays. But I said at the beginning that serious Perl users will need to go on to consult more comprehensive manuals than this brief beginners' textbook. Some experts would argue that even beginners need to learn about references. I believe it will be better to leave them aside, until the reader has grown in confidence working with the Perl constructions that this book does cover. From now on, then, we shall pretend that Perl does have true arrays of arrays, but we shall be careful always to put values into arrays of arrays, and get values out, one individual cell at a time.

Download free ebooks at bookboon.com

That is a constraint on the arrays-of-arrays concept in Perl. In another respect, though, Perl arrays of arrays are *more* flexible than in some other programming languages. Our array `@counties`, although it is composed of arrays rather than of scalars (really, "appears to be composed …" – but I shall stop saying that, and write from now on as if appearance were reality), nevertheless is what one might call "regular". The value in *each* row is always an array containing exactly two elements, one string and one number. In some languages, multi-dimensional arrays *must* be regular. But nothing in Perl requires this. It is perfectly possible to create an array in which rows contain different numbers of elements. (It is even possible to create an array in which some rows contain individual scalars, other rows contain arrays with different numbers of elements, and some of those elements could in turn themselves be arrays.)

How often a thoroughly chaotic array like that last one would actually be useful in practice is another question. Very rarely, I imagine. But mildly irregular arrays often are useful. Although we have introduced the concept "array of arrays" through an example which is totally regular, bear in mind that this is just a special case.

Please click the advert

Download free ebooks at bookboon.com

16 User-defined functions

16.1 Adapting Perl to our own tasks

In chapter 7 we saw that Perl includes dozens of built-in functions, such as `sqrt()`, `substr()`, `rand()`, and we examined a few of them. But often we need our program to execute some specific function that does not feature on the list of built-in functions, perhaps because it relates too closely to the particular task we are working on. In that case, we will need to create our own *user-defined function*.

For instance, in our code-chunk (18) to read in a file of county population data, one of the actions was to replace any sequence of one or more whitespace characters internal to a string with a single underline character, so that a string of the form, say,

```
Cambridgeshire and   Isle of Ely
```

is turned into:

```
Cambridgeshire_and_Isle_of_Ely
```

In our program this was handled by a single line, line 18.14:

```
$name = join("_", split(/\s+/, $name));
```

But in a larger program, which perhaps has to read various kinds of data in from several different files, there might be numerous situations where we want to make names more "machine-friendly" by carrying out this same routine of changing internal whitespace to single underline characters. We could copy 18.14 at every point in the program where this routine needs to be executed, replacing `$name` with whatever variable the routine needs to be applied to at the point in question. But 18.14 is a complicated thing to type out (even in its newer form – in its original form as line 12.14 of program (12) it was even more fiddly); it is inefficient and invites errors to repeat such details at different places in a program.

What we need to do is define a function, call it `despace()`, which takes a string as argument and does to it what 18.14 does to the string `$name`. Once the `despace()` function is defined, we can simplify (18) by replacing 18.12 (which assigns the value of the temporary variable `$1` to `$name`) and 18.14 (which, as we have seen, replaces whitespace by underlines in `$name`) with the single line:

```
$name = despace($1);
```

and, wherever else in a longer program we need to alter character strings in that particular way, we can do so using a similarly concise statement.

Download free ebooks at bookboon.com

16.2 The structure of a user-defined function

So how do we define the `despace()` function? Like this:

(19)

```
sub despace
  {
  my $p = shift(@_);
  $p = join("_", split(/\s+/, $p));
  return($p);
  }
```

The keyword `sub` stands for "subroutine"; the terms "function" and "subroutine" are interchangeable in Perl.[26]

It is logical to keep function definitions separate from the main sequence of statements that a program executes when it is run, so commonly a function definition like this would appear either at the beginning or at the end of the program. But Perl does not actually care where a user-defined function definition appears; whenever Perl reaches a statement which *calls* that function – that is, a statement such as `$name = despace($1)`, whose meaning depends on the definition of `despace()` – then Perl will find the relevant function definition and use it, so long as it exists in the program file somewhere.

Please click the advert

Download free ebooks at bookboon.com

What the `despace()` definition is essentially saying is this: "Whatever the argument of this function is actually called, for the purposes of defining the function it will be called `$p`. Now `$p` is changed in the way that line 18.14 changed the string `$name`. And after that has been done, the new version of `$p` is delivered back to the code which called the function, as the result *returned* by the function call." The keyword `return` identifies what follows it as the result of that function call.

The only new complication in this relates to line 19.3, `my $p = shift(@_)`. Bear in mind that at the time we write the function definition, we don't know what its argument will be named when the function is used. In (18) the string which had its spaces changed to underlines was called `$name`, but on another occasion we might want to apply `despace()` to a string called `$a123`, or `$broccoli`, or anything. So we need an arbitrary name internal to the function definition to stand for the real name that appears in the calling code – any arbitrary name would do, I happened to choose `$p`. Line 19.3 is what tells Perl that the role of `$p` in the function definition is to represent the argument.

Perl does this in a more roundabout fashion than some programming languages. If Perl were more "ordinary" in this respect, line 19.1 might run `sub despace($p)` – announcing the name to be used for the argument at the same time as the function name `despace` is introduced. In Perl that does not work. Instead, whenever any function is called, whatever argument(s) it is called with are assigned to a special array name `@_`. (Remember that a function may have more than one argument; this applies to user-defined functions as well as to built-in functions.) In the case of `despace()` there is only one argument, so `@_` will be a one-element array; and `shift()` will give us the first element (in this case, the only element) of the `@_` array. Line 19.3 assigns that argument to a name `$p`, under which name we are free to manipulate it.

The remaining feature of line 19.3 which needs explanation is the word `my`. This word before a variable name makes the variable *local* rather than *global*. If we are writing a large program, as well as many lines of "top-level" code outside any subroutine, there may be numerous different subroutines – indeed it is likely that some complex subroutines will themselves call lower-level subroutines, and so on down. While we are defining a subroutine needed for some quite separate aspect of the program, we will probably have forgotten all about the details of `despace()`, and we could easily decide to use the variable name `$p` for some unrelated purpose in that other subroutine. Unless we explicitly declare `$p` as local to `despace()`, the same name `$p` appearing in another subroutine will refer to the same variable, and things done in one subroutine might interfere with what is intended to happen in the other. Variables are global by default, but we want `$p` in `despace()` to be local, so that it is treated by Perl as independent of any variables that happen to share the same name in other parts of the program. The word `my` achieves that.

Line 19.3 declared `$p` as local at the same time as `$p` was assigned a value, but a `my` statement can be a separate statement; we could have written:

```
my $p;
$p = shift(@_);
```

If our subroutine needs several local variables, we can declare them all in a single line within brackets:[27]

```
my ($p, $q, $r);
```

The *scope* within which a variable is local does not have to be a subroutine, but that is probably the most usual way to use `my` and the only one we shall discuss.

Returning to our definition of `despace()`: once 19.3 has defined `$p` as standing for the argument, 19.4 then applies exactly the same substitution operation to `$p` as line 18.14 applied to `$name`. Finally, the word `return` in 19.5 defines what the function delivers to the calling code as its result – in this case, `$p` as it exists after 19.4 has applied to it.

So code-chunk (18) will now be replaced by (20), in which the main part of the program contains a line 20.12 which calls the function `despace()` with the string `$1` captured by line 20.6 as argument, and the result of applying `despace()` to `$1` becomes the value of `$name`:

(20)

```
1   open(INFILE, "<../data/pops/countyData.txt")
      or die "Cannot open input file\n";
2   $i = 0;
3   while ($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 = despace($1);
13      $population = $2;
15      $population *= 1000;
16      $counties[$i][0] = $name;
17       $counties[$i][1] = $population;
18       ++$i;
19      }
20    }
21  close(INFILE);

22  sub despace
23    {
24    my $p = shift(@_);
25    $p = join("_", split(/\s+/, $p));
26    return($p);
27    }
```

16.3 A second example

The function `despace()` is one that applies to strings. As further practice with the concept of user-defined functions, let's now define a function that works with numbers.

In chapter 7 we saw that the built-in function `int()` turns decimals into whole numbers in a crude fashion, which does not give the nearest whole number if the fractional part of the argument is half or more. It could often be useful to have a function `round()`, which always takes a decimal to the nearest whole number, rounding up or down as appropriate. This function is easy to define.

If you think about it, one way to get the right answer would be to add one-half (0.5) to the number we want to round, and then apply the built-in `int()` function to the result. We want e.g. 3.14 to be rounded to 3: 3.14 plus one-half is 3.64, and `int(3.64)` is 3; but we want e.g. 3.75 to be rounded to 4: 3.75 + 0.5 = 4.25, and `int(4.25)` gives 4. Well, actually, this is not quite the whole story, because we haven't taken negative numbers into account yet. If the argument to `round()` is negative, we want to *subtract* one-half before applying `int()`, because we want `round(-3.14)` to be –3 and `round(-3.75)` to be –4.

Please click the advert

The financial industry needs a strong software platform
That's why we need you

SimCorp is a leading provider of software solutions for the financial industry. We work together to reach a common goal: to help our clients succeed by providing a strong, scalable IT platform that enables growth, while mitigating risk and reducing cost. At SimCorp, we value commitment and enable you to make the most of your ambitions and potential.

Are you among the best qualified in finance, economics, IT or mathematics?

Find your next challenge at
www.simcorp.com/careers

SimCorp

www.simcorp.com
MITIGATE RISK | REDUCE COST | ENABLE GROWTH

Download free ebooks at bookboon.com

So here is a function which will give the correct result with both positive and negative arguments:

(21)

```
1   sub round
2     {
3     my $n = shift(@_);
4     return($n >= 0 ? int($n + 0.5) : int($n - 0.5));
5     }
```

Line 20.4 uses the `?:` construction to ask "Is `$n` positive? If yes, return the result of applying `int()` to `$n` plus 0.5; if no, return the result of applying `int()` to `$n` minus 0.5."

16.4 Multi-argument functions

Functions such as `despace()` or `round()`, which take a single argument, are the simplest case. To broaden our grasp of user-defined functions, let's look at one taking several arguments. Perhaps we are interested in checking the average length of collections of words used in some circumstance, and we want to count only wholly-alphabetic words, not "words" composed partly or wholly of numerals. So we want to define a function `aveWordLen()` which gives results as follows:

```
print aveWordLen("new", "sale", "special", "executive");
```

```
5.75
```

```
print aveWordLen("leading", "99p", "bargain");
```

```
7
```

(The words *leading* and *bargain* both have seven letters, and *99p* will be ignored because it contains digits.)

Notice that `aveWordLen()` does not merely take multiple arguments; it can take any number of arguments on different occasions. Since an array can be any length, the `@_` system means that this is no special problem.[28]

Here is a definition for `aveWordLen()`:

(22)

```
1   sub aveWordLen
2     {
3     my $total = 0;
4     my $N_words = 0;
5     my $word;
6     foreach $word (@_)
7       {
8       if ($word !~ /\d/)
9         {
10        ++ $N_words;
11        $total += length($word);
12        }
13      }
14    return($total / $N_words);
15    }
```

Line 22.6 assigns each string in the argument array successively to the local variable `$word`; 22.8 checks that `$word` contains no digit, and (provided it passes the check) lines 22.10–11 increment `$N_words` by one and `$total` by the length of `$word`. Line 22.14 defines the result as the total of all word lengths divided by the number of words.[29]

16.5 Divide and conquer

It is clear that if a function like `aveWordLen()` is needed at several places in a program, it is much better to define it once and call it by name whenever it is needed, than to copy a long chunk of code such as the one above into all the different places where it is needed. It is not just a matter of reducing the overall length of the code – that in itself does not matter much (though the more lines, the more opportunity for typing mistakes). But more important is the principle of "divide and conquer". When a program is required to execute a complex range of processes, the programmer is much more likely to be able to get it right if the overall functionality can be broken down into many small pieces, each of which can be coded independently of the others – so that only a limited number of issues have to be held in the programmer's head at any one time. User-defined functions and local variables are a large part of the machinery which Perl, and other languages, use to facilitate the "divide and conquer" approach.

Consequently, an important part of learning to program well is learning to notice cases where parts of a longer process can be separated out and coded as subroutines. It is rarely a good idea for a program to contain a lengthy "main" section (the part of the code which the machine begins to execute when the program is run, outside all the `sub`'s). Good, robust software is commonly articulated into many separate functions, so that a fairly short main section contains several function calls (and the functions will often call further functions, and so on).

16.6 Returning a list of values

Each function we have defined so far has returned a single item (a scalar). But we can also define a function to return a list. Let's define a function `maxima()` which uses our reformatted file of county population data to identify both the largest county population and the longest county name. This is perhaps not a very realistic task in practice – why would we want to investigate both of those two issues simultaneously? – but it makes a good exercise. (Tasks that are useful for teaching programming concepts often are unrealistic.)

We'll say that in the main part of our program, outside the subroutines, there will be lines which simply ask for the two maximum figures and incorporate them in messages to be printed out, like this:

(23)

```
@figures = maxima();
print ("Largest county pop. is $figures[0]\n");
print ("Longest county name is $figures[1]\n");
```

```
Largest county pop. is 7,914,000
Longest county name is Huntingdonshire_and_Peterborough
```

The data will be taken from the reformedData file created by program (12); and all the work of reading these data in and working out which entries are the respective maxima will be delegated to the `maxima()` function.

Remember that in the reformedData file each line contains a population figure with commas grouping digits into threes, followed after a tab by a county name, e.g.:

```
353,000 Cornwall
```

We shall be doing arithmetic with the population figures, to discover which is largest, so `maxima()` will need to remove the commas before comparing one figure with another (we know that Perl does not recognize a string such as `353,000` as a number). We can handle this with a further user-defined function, `decomma()`, which takes a string as argument and returns the string with any commas stripped out and the gaps closed up. And when returning the maximum population as a figure ready to be displayed to the user, it would be nice to have the commas put back in (as in the output shown in italics above); so we shall write another function, `recomma()`, which will use the final version of line 12.16 (see section 11.3 above) as a routine for inserting commas in the correct places.

Here are the three functions. (I have left a blank line between them, and in typing out a program one normally would leave some space between separate subroutines, simply for readability; Perl does not care whether the blank line is there or not.)

Download free ebooks at bookboon.com

(24)

```
sub maxima
  {
  open(INFILE, "../data/pops/reformedData.txt") or
    die("Can\'t open reformedData.txt\n");
  my($longestName, $largestPop, $line, $name, $pop);
  $longestName = "";
  $largestPop = 0;
  while ($line = <INFILE>)
    {
    chomp($line);
    if ($line !~ /^(\S*)\t(\S*)$/)
      {
      die("input line in unexpected format: $line\n");
      }
    $pop = decomma($1);
    $name = $2;
    if (length($name) > length($longestName))
      {
      $longestName = $name;
      }
    if ($pop > $largestPop)
      {
      $largestPop = $pop;
      }
    }
  close(INFILE);
  return(recomma($largestPop), $longestName);
  }

sub decomma
  {
  my $numeral = shift(@_);
  $numeral =~ s/,//g;
  return($numeral);
  }

sub recomma
  {
  my $popFig = shift(@_);
  while ($popFig =~ s/(\d)(\d{3})($|,)/$1,$2$3/)
    {;}
  return($popFig);
  }
```

The function `maxima()`, which is a function taking no arguments, begins (24.5–6) by setting the variables `$longestName` and `$largestPop` to the shortest possible name (the length-zero "null string") and smallest possible population (zero) respectively, and then looks at the name and population on each line of the input file. Since we are taking our input data from a file we created ourselves, it should be safe to assume that each line has the numeral + tab + name format – but just in case the reformedData file has somehow become corrupted, 24.10 arranges for the program-run to terminate with an appropriate error message if a line with a different format is encountered. For each population figure read in, 24.14 in the `maxima()` function calls the `decomma()` function to change it into a form it can do arithmetic with. Whenever `maxima()` finds a name or a population larger than the previous maximum found, the new name or population replaces the current value of `$longestName` or `$largestPop` respectively as current maximum (24.16–23). The `maxima()` function returns the overall maxima as a list with two members, within which the population maximum is in the form produced by the `recomma()` function; in (23), within the main part of the program, this two-member list is assigned as the value of an array `@figures`.

Thus in (23) the main part of the program calls `maxima()`, which in turn calls `decomma()` and `recomma()`; the return values of `decomma()` and `recomma()` contribute to calculating the return value of `maxima()`, and that return value in turn contributes to executing the statements in (23). This is how real-life Perl programs will typically be structured. The main part of the program, outside any `sub` definition, may be quite short, but it will contain function calls; the subroutines they call will often themselves contain calls to further functions, and there may be further levels below levels of user-defined functions until one eventually gets down to functions that use only built-in Perl constructions.

Please click the advert

Download free ebooks at bookboon.com

16.7 “Subroutines” and “functions”

Those readers who have learned other programming language(s) before encountering Perl might be surprised that Perl uses the keyword `sub` to define functions, and that I have been using the terms “function” and “subroutine” interchangeably – as mentioned earlier, for languages other than Perl, subroutines and functions are two different things. A subroutine *does* something – it carries out an action, for instance it changes the values of its arguments in some systematic way, displays the arguments in some format, or the like; a function calculates a value and returns it for use by another part of the program, commonly without changing anything. And this difference in purpose is reflected, in many programming languages, by different notation conventions.

In Perl there is no difference between subroutines and functions. All the user-defined functions we have looked at so far contained a `return` statement returning a value to the part of the program which called the function; but a Perl function does not have to contain a `return` statement. Here is a function which takes a string as argument and prints out its words (elements separated by whitespace) on separate lines, in other words it carries out an action rather than returning a value:

(25)

```
1   sub printlnbyln
2     {
3     my $string = shift(@_);
4     my @words = split(/\s+/, $string);
5     my $word;
6     foreach $word (@words)
7       {
8       print($word, "\n");
9       }
10    }
```

A program containing this function definition could have a statement like:

```
printlnbyln("the quality of mercy");
```

```
the
quality
of
mercy
```

Since `printlnbyln()` contains no `return` statement, one would not normally use a statement like:

```
$a = printlnbyln("the quality of mercy");
```

(In fact Perl is organized so that even subroutines without `return` statements do deliver return values; but this is one of the Perl wrinkles that is best left to the geeks. Learners are advised only to use return values from defined functions where the return values are explicitly identified as such by `return` statements.)

We saw in chapter 9 that while most Perl built-in functions return values, some, such as `chomp()`, exist mainly in order to change their arguments. Likewise a code block headed by a `sub` line can be designed to carry out an action, it can be designed to return a value, or it can be designed do both of these. There is no distinction between subroutines and functions, so it is reasonable to use `sub` as a keyword for user-defined "functions".

Please click the advert

RedStarRésumé

Do you want your Dream Job?

More customers get their dream job by using RedStarResume than any other resume service.

RedStarResume can help you with your job application and CV.

Go to: Redstarresume.com
Use code "BOOKBOON" and save up to $15

(enter the discount code in the "Discount Code Box")

Download free ebooks at bookboon.com

17 Hash tables

17.1 Tables indexed by strings

An array is a table of data where the rows are identified by their numerical position in sequence. A *hash table* (or "hash" for short), with `%` rather than `@` prefixed to its name, is like an array but the rows are indexed not by numbers but by character-strings. With a hash, it does not make sense to talk about "the sixth row" or "the initial row"; but, if we chose to use, say, names of fish as the *keys* indexing the rows of a hash, then we could ask what was in the "pike" row or the "mackerel" row.

(The values *within* the rows of a hash table, like those in the rows of an array, can be any items you like, numbers or strings; it is the way that rows are identified which creates the contrast between arrays and hashes.)

In an introductory textbook the main difficulty about hashes is not explaining the mechanics of how they work, but making clear what their point is – why anyone would want to use them. For practical reasons, the programming examples included in textbooks are normally little "toy" examples; but hashes are most useful when they contain not just a dozen or two dozen data items but hundreds, or thousands, of rows of data.

So let's consider an example like that. We shan't be able to inspect the entire contents of the hash, but we will see how it is created and, once it exists, how it is used; and because it is a large enough example to be realistic we will be able (I hope) to see why hashes are a useful thing to have available. Along the way we shall encounter one or two other, much simpler but important features of Perl that we happen not to have met yet.

As background to our hypothetical hash table, let us suppose that we are interested in the vocabulary of advertising copy. Perhaps we want to monitor trends in the choice of words used in marketing, so that we can make our own advertising come over as fresh and up-to-date, or so that we can draw links between the language of advertising and other social trends. Never mind why exactly we want to study this topic; the point is that we will suppose we have files containing a mass of examples of advertising copy, and we want to process these into a table which, for any word found in the files, gives a count of the number of occurrences of that word.[30]

The table will be a hash table; we'll call it `%adWords`. The words themselves will be the "keys" indexing the rows of the table; the contents of a row will be the frequency within our data of the word which is the key of the row. So for instance the *bargain* row might contain the value 546, meaning that the word *bargain* occurred 546 times in the files from which the table was compiled.

Suppose our program has created a table like `%adWords`. Then the way to extract an individual data item such as the one just mentioned will be to use curly brackets round the key, corresponding to square brackets round a row-number in the case of an array:

```
print $adWords{"bargain"};
```

```
546
```

The data item is a scalar, so a dollar sign appears at the beginning of `$adWords{"bargain"}` even though `%adWords` as a whole takes the `%` prefix. By now, this is what we expect in Perl.

Notice that there is no way that we can ask for, say, the contents of row 13 of `%adWords`. If we wrote `$adWords{"13"}` that would give us the frequency of the word *13*; and `$adWords{13}`, or `$adWords[13]`, would just be errors. We shall see, later, how it is possible to work through each row of a hash table in turn; it is not done by starting with row 0 and moving on to row 1, row 2, etc. – with a hash these phrases are meaningless. Hashes are useful in situations where, much of the time, one is looking at single data items within a large table, rather than working through the entire table.

17.2 Creating a hash

So how would we create the `%adWords` hash from our text files?

To save time I'll assume that we have arranged a filehandle so that the expression `<INFILE>` will read a line of text in from our data files, until there is nothing left to read in. After reading in any line, we will want to split it into an array of words separated by spaces (an array, not a hash); and then we'll work through the array using the individual words to increment the values in the corresponding rows of `%adWords`.

Of course, if we split a line of everyday text on whitespace, the units we get will not all be "words" exactly – some of them might have punctuation at beginning or end (or both), and indeed some elements surrounded by whitespace might consist wholly of punctuation (dashes are often written with spaces on either side). We don't want, say, `bargain!` or `"bargain` (or `Bargain`) to be counted separately from `bargain`, so let's say that before adding any word into the hash we'll remove any punctuation found at either end (and ignore a "word" which contains only punctuation marks), and we'll reduce all alphabetic characters to lower case. Also, as with `avewordLen()` in chapter 16, we'll ignore words containing digits – any number can appear in a price, such as *99p* or *£279*, so it would not be very meaningful to keep count of the prices that happen to occur in our data.

Download free ebooks at bookboon.com

The following code will do to read in the lines of data and get the words ready to add into the hash table; line 26.11 does the work of actually building the hash table with the help of a user-defined function, which we shall look at below.

(26)

```
1   while ($line = <INFILE>)
2     {
3     @words = split(/\s+/, $line);
4     foreach $word (@words)
5       {
6       if ($word =~ m/\w/ and $word !~ m/\d/)
7         {
8         $word =~ s/^\W*(\w)/$1/;
9         $word =~ s/(\w)\W*$/$1/;
10        $word = lc($word);
11        updateAdWords($word);
12        }
13      }
14    }
```

Please click the advert

Try this...

All Options

The sequence 2, 4, 6, 8, 10, 12, 14, 16, ... is the sequence of even whole numbers. The 100th place in this sequence is the number...?

Challenging? Not challenging? Try more ▸▸ www.alloptions.nl/life

Download free ebooks at bookboon.com

Line 26.3 splits an input line into words separated by one or more whitespace characters. Then 26.4–14 work through each element of the resulting array in turn, using `$word` to refer to the current array item. Line 26.6 checks that `$word` contains at least one "word character" and no digit – if it fails these tests, for instance because it is all punctuation, nothing further will happen in this pass of the `foreach` loop (and Perl will move on to consider the next word). Line 26.8 deletes any sequence of nonword characters at the beginning of `$word`, and line 26.9 does the same at the end of `$word`. Line 26.10 reduces any capitals in the remaining string to lower case. Then 26.12 calls the subroutine `updateAdWords()`, which uses `$word` to update the `%adWords` hash table.

When a word is used to update `%adWords`, there are two possibilities: either this is a word which `%adWords` has not seen before, in which case a row needs to be created for it in the hash table with the value 1; or there is already a row for that word, in which case the value in that row needs to be increased by one. In the definition of the function it is simpler to take those alternatives in the opposite order, using the built-in function `exists()` which checks whether a given hash contains a row with a given key:

(27)

```
1   sub updateAdWords
2     {
3     my $entry = shift(@_);
4     if (exists($adWords{$entry}))
5       {
6       ++ $adWords{$entry};
7       }
8     else
9       {
10      $adWords{$entry} = 1;
11      }
12    }
```

(Strictly speaking, there is a third possibility: the first time `updateAdWords()` is called, the hash `%adWords` will not yet exist. But as soon as Perl hits 27.10 for the first time, it will create `%adWords` without further ado, as well as creating a row for the current word within it.)

And that is all there is to it, so far as creating the hash table is concerned. After (26) has run through all the data so that `<INFILE>` has no further lines of text to deliver, `%adWords` will have a row for each distinct word in the data; the key of the row will be the word and the value will be the number of times that word occurs.

If we want to delete an individual entry from a hash table, the built-in function `delete()` does this:

```
delete $adWords{"bargain"};
```

You might perhaps wonder why hashes have a `delete()` function but there is no equivalent for arrays. What would an equivalent do in the case of an array, though? When a table is indexed by row numbers, one cannot just make row 15 (say) go away – we can remove the contents of the row, but the row will still be there between rows 14 and 16. With a hash, on the other hand, after the command `delete $adWords{"bargain"}` is executed, it is as if `%adWords` had never had a `bargain` row in the first place.

17.3 Working through a hash table

We saw above that the typical use of a hash is to extract the value for an individual key; and we do that by placing the key within curly brackets. But, sometimes, we do want to work through every entry in a hash table in turn. As an exercise, we might add up all the values in `%adWords` entries in order to get a count of the total number of words (word-tokens) in our data files.

We cannot apply `foreach` directly to a hash; `foreach` applies only to arrays. For hashes, there is a related term `each`. But `each` does not mean quite what you might expect; its usage is rather counterintuitive. If `%hash` is a hash table, the expression `each(%hash)` really means "the next key/value pair in `%hash`, if there is another one". So we can go about totalling the values in `%adWords` this way:

(28)

```
1  $total = 0;
2  while (($key, $value) = each(%adWords))
3    {
4    $total += $value;
5    }
6  print "$total\n";
```

Line 28.2 takes successive key/value pairs from `%adWords` and makes the members of the pair the values of the variables `$key` and `$value`; `$key` is not needed, but `$value` is used to increment `$total`.

In 28.2, assigning an array (in this case the two-element array returned by `each`) to a list of variables within round brackets is a neat way of making multiple assignments with one equals-sign. We could alternatively have invented a variable name to contain the successive key/value pairs. That is, we could have written 28.2 as, say, `while (@kvpair = each(%adWords))`, in which case line 28.4 would have needed to become `$total += $kvpair[1]` (i.e. the second element of `@kvpair` – the key is `$kvpair[0]`). But the way we have done things in (28) seems clearer; there is no point in turning the key/value pair into an array with a name of its own, rather than a nameless list, unless that array will later be used in ways that it is not used here.

There are other ways to work through the full set of entries in a hash table. Thus, the built-in functions `keys()` and `values()` return arrays containing respectively all the keys, and all the values, in a hash named as argument. Since the result of `values()` is an array, we can use `foreach` with it:

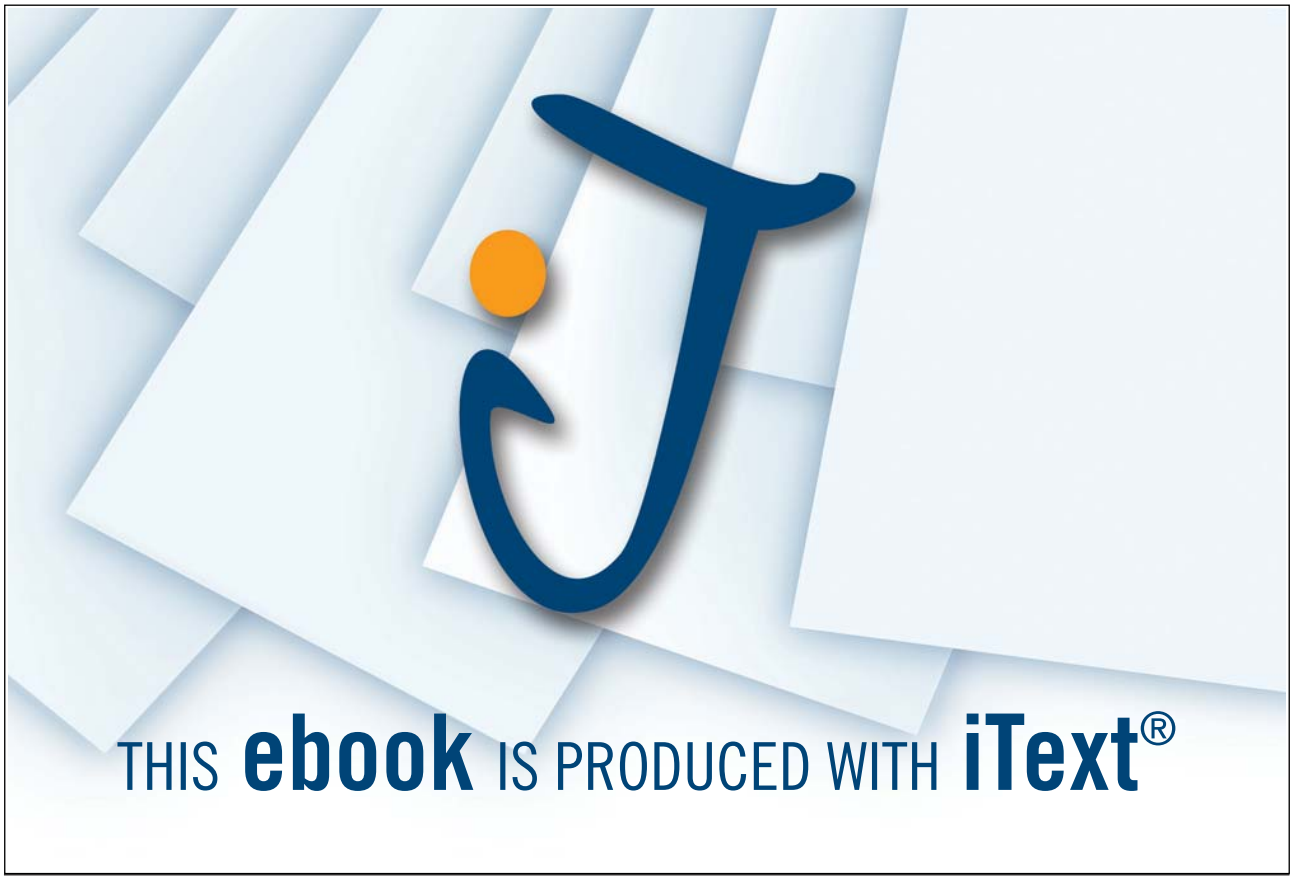
(29)

```
1  $total = 0;
2  foreach $value (values(%adWords))
3    {
4    $total += $value;
5    }
6  print $total;
```

17.4 Advantages of hash tables

We have seen that `each` returns successive key/value pairs from a hash, and `keys()` and `values()` return the respective items as an array, which is a sequentially ordered structure. So it may seem that I have misled readers by saying that there is no concept of "position of a row in numerical sequence" for hash tables. In what order are items delivered by `each`, `keys()`, and `values()`?

Please click the advert



Download free ebooks at bookboon.com

The answer is, to all intents and purposes in a random order. Inside the computer the rows of a hash table must be in some order or other, but there will not be any obvious logic to the order. In particular, the order of the rows *will not reflect the sequence in which entries were added to the hash*.

Normally, this book does not "peer under the bonnet" to discuss how programming structures are actually implemented within the machine. At this level, to do that would commonly be confusing rather than helpful. In the case of hash tables, though, it may be worth making an exception and explaining just a little about how they are implemented and why they store elements in an apparently random sequence, in order to help readers appreciate the "point" of including hashes in a programming language.

If a hash table is large, the process of looking up individual items in it is much more efficient than in the case of an array. To look up an item in an array, the machine must begin at row zero and inspect successive rows until the answer is found – on average half of the entire array has to be checked. In a hash, an item is placed in a row having some arithmetic relationship to its key. For instance, if the hash has 1000 rows available, then to decide which row to use for key *X* the rule could be to multiply together the ASCII codes for the characters of *X* (a = 97, b = 98, etc.) and to use the last three digits of the large resulting number to place the item – or, if that particular row happens already to be taken for a different key, use the next empty row. Then, when an item is looked up, by making the calculation on the key string the machine can usually go straight, or almost straight, to the correct row even in a huge table. For many real-life applications, this is an important benefit.[31]

Efficient lookup is probably the most important reason for including hash tables in a programming language. But also, it can be easier for a programmer to think in terms of tables indexed by meaningful strings than tables indexed by numbers. Consider again our table of county data. At present, the only datum included for a county, apart from its name, is its population. But we might want to enlarge the table to include further numerical information – say, the area of the county in acres, and its property-tax base in pounds sterling (what back in 1966, the year to which our figures refer, was called "rateable value"). As a table on paper, its first few lines would look like this:[32]

	population	acreage	rateable value (£)
Bedfordshire	427,970	305,089	12,789,675
Berkshire	585,450	463,830	19,770,843
Buckinghamshire	542,020	477,750	29,709,579

Figure 2

Let's suppose we have these data in an electronic file `countyDataPlus.txt`, in a format having one line per county, no commas in the numbers, and fields separated by the "|" symbol, possibly with adjacent whitespace.[33] In other words the first line of the file might look like this:

```
Bedfordshire | 427970|305089| 12789675
```

Then we can read the data into a Perl program using code such as:

(30)

```
1  while ($line = <INFILE>)
2    {
3    chomp($line);
4    $line =~ s/^\s*(\S.*\S)\s*$/$1/;
5    @items = split(/\s*\|\s*/, $line);
6    ($name, $population, $acreage, $rateable_value) = @items;
7    $name = despace($name);
       ⋮
     }
```

Line 30.4 will remove any whitespace from beginning and/or end of the line read in, 30.5 will break it up into an array of strings separated by the vertical bar possibly with adjacent whitespace, 30.6 will assign the respective elements of the array to individual scalar variables, and 30.7 will modify multi-word county names using the `despace()` subroutine, (19), to replace internal whitespace by single underline characters.

But now we have to decide how to arrange these items into a data-structure within our program, so that later parts of the program can look the information up.

Probably we shall still want to implement the table as an array, rather than a hash with county names as keys – it would not be specially convenient to use a string like `Cambridgeshire_and_Isle_of_Ely` for lookup purposes. But, now we know about hashes, we can choose whether we want the table to be an array of arrays, or an array of hashes. If it is an array of arrays, then the rows are indexed by integers 0 to 45 and the columns by integers 0 to 3:

```
print "$counties[1][0]\n";
```

Berkshire

```
print "$counties[1][2]\n";
```

463830

If on the other hand we go for an array of hashes, each row of the array will be a little four-element hash, which could be indexed with the keys `name`, `pop`, `acreage`, `rValue`:

```
print "$counties[1]{name}\n";
```

Berkshire

```
print "$counties[1]{acreage}\n";
```

463830

(Provided our hash keys are all-letter strings like `name` or `acreage`, we don't need to surround them with quotation marks inside the curly brackets – though if, for instance, we used a two-word string with a space in the middle as a key, this would need quotation marks.)

With only four elements, the lookup efficiency of a hash is irrelevant. But it takes a burden off the programmer if he does not need to remember that acreage is in column 2, and so on. Some Perl programmers would argue that this is really the central advantage of hashes, more significant in practice than the point about efficiency of lookup in large tables.

Please click the advert

Masters in Management

Designed for high-achieving graduates across all disciplines, London Business School's Masters in Management provides specific and tangible foundations for a successful career in business.

This 12-month, full-time programme is a business qualification with impact. In 2010, our MiM employment rate was 95% within 3 months of graduation*; the majority of graduates choosing to work in consulting or financial services.

As well as a renowned qualification from a world-class business school, you also gain access to the School's network of more than 34,000 global alumni – a community that offers support and opportunities throughout your career.

For more information visit www.london.edu/mm, email mim@london.edu or give us a call on +44 (0)20 7000 7573.

* Figures taken from London Business School's Masters in Management 2010 employment report

London Business School

Download free ebooks at bookboon.com

17.5 Hashes versus references to hashes

The same point we saw earlier about arrays of arrays only *seeming* in Perl to be arrays of multi-item elements applies equally to arrays of hashes (and to hashes of arrays, hashes of hashes, ...). So the code which populates our enlarged `@counties` array with data will need to insert values one cell at a time:

```
$counties[$i]{name} = $name;
$counties[$i]{pop} = $population;
$counties[$i]{acreage} = $acreage;
$counties[$i]{rValue} = $rateable_value;
```

That is, after the `open` statement which makes INFILE the handle for our file of data, the program will run like this:

(31)

```
1   $i = 0;
2   while ($line = <INFILE>)
3     {
4     chomp($line);
5     $line =~ s/^\s*(\S.*\S)\s*$/$1/;
6     @items = split(/\s*\|\s*/, $line);
7     ($name, $population, $acreage, $rateable_value) = @items;
8     $name = despace($name);
9     $counties[$i]{name} = $name;
10    $counties[$i]{pop} = $population;
11    $counties[$i]{acreage} = $acreage;
12    $counties[$i]{rValue} = $rateable_value;
13    ++$i;
14    }
15  close(INFILE);
```

We could replace 31.7–8 with code that turns the items found in the current input line into a four-element hash (using the symbol `=>` to link key/value pairs in a list), like this:

```
%countyHash = (name => despace($name), pop => $population, acreage
   => $acreage, rValue => $rateable_value);
```

but, if we were to do that, we could not go on to replace 31.9–12 by a single statement:

```
%counties[$i] = %countyHash;   #WON'T WORK!
```

Likewise, if we need all the data about a given county, we shall need to extract it from the hash one item at a time:

```
$name = $counties[$i]{name};
$population = $counties[$i]{pop};
   etc.
```

But, provided we respect this constraint, we can forget that appearance is different from reality. (And, once you go on in due course to learn about references, you will be able to escape from the constraint.)

Please click the advert

You're full of *energy and ideas*. And that's just what we are looking for.

Looking for a career where your ideas could really make a difference? UBS's Graduate Programme and internships are a chance for you to experience for yourself what it's like to be part of a global team that rewards your input and believes in succeeding together.

Wherever you are in your academic career, make your future a part of ours by visiting www.ubs.com/graduates.

www.ubs.com/graduates

UBS

© UBS 2010. All rights reserved.

Download free ebooks at bookboon.com

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
```

Download free ebooks at bookboon.com

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:

Download free ebooks at bookboon.com

(32)

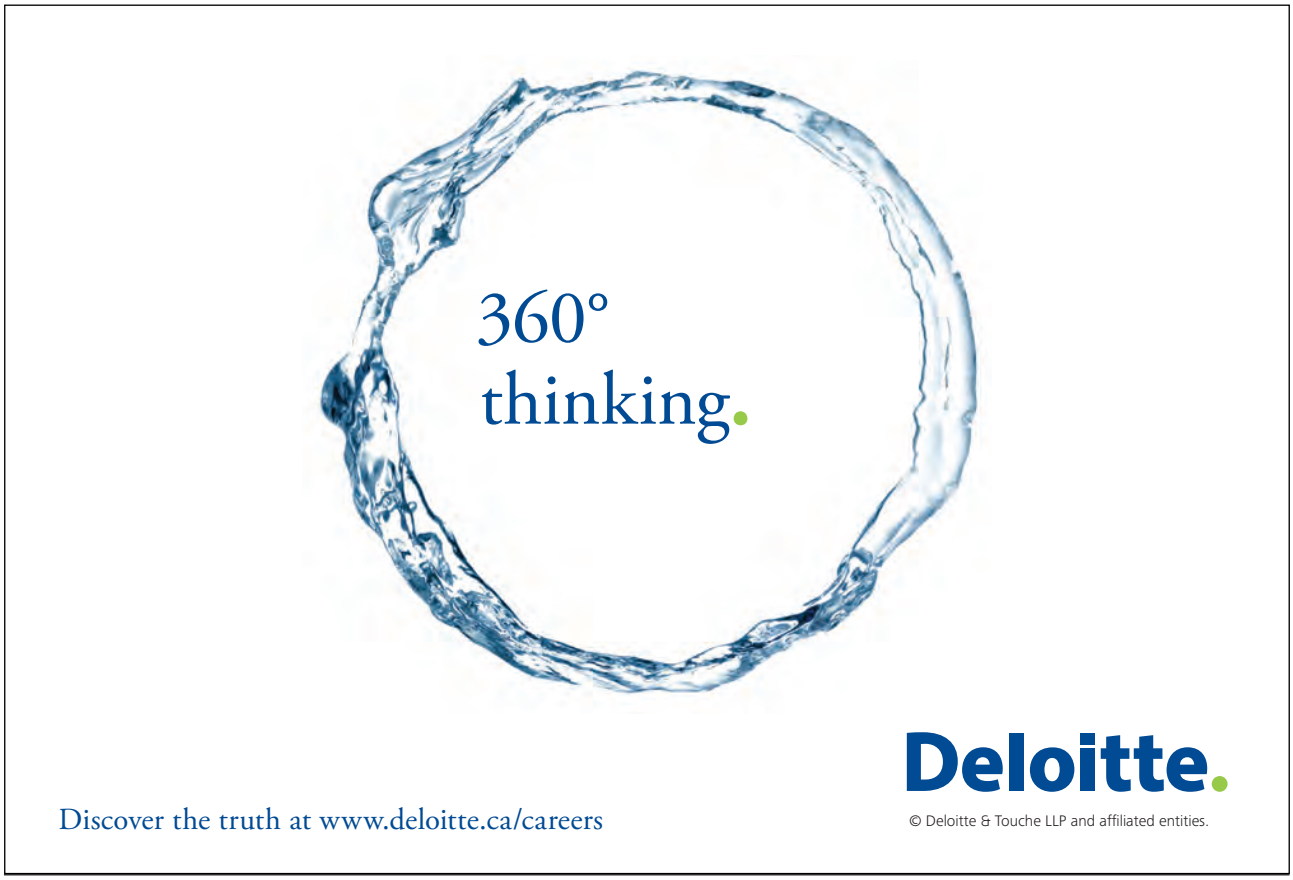
```
1   for ($j = 0; $j < @counties; ++$j)
2     {
3     printf("Pop. density of %s is %.3f people per acre\n",
        $counties[$j]{name},
        $counties[$j]{pop}/$counties[$j]{acreage});
4     }

    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.

Please click the advert



Download free ebooks at bookboon.com

(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!)

Download free ebooks at bookboon.com

(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.

Please click the advert

Download free ebooks at bookboon.com

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)

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

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

```
1277200878
6
```

Please click the advert

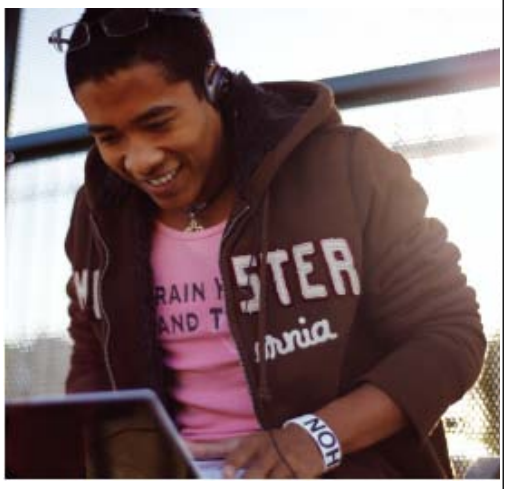
ericsson.com

YOUR CHANCE TO CHANGE THE WORLD

Here at Ericsson we have a deep rooted belief that the innovations we make on a daily basis can have a profound effect on making the world a better place for people, business and society. Join us.

In Germany we are especially looking for graduates as Integration Engineers for
- Radio Access and IP Networks
- IMS and IPTV

We are looking forward to getting your application! To apply and for all current job openings please visit our web page: **www.ericsson.com/careers**



ERICSSON

Download free ebooks at bookboon.com

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.

Download free ebooks at bookboon.com

20 The debugger

Anyone who writes programs to carry out realistic, non-trivial tasks is very familiar with the fact that they do not often work properly first time. Without noticing it, we make mistakes in translating our intentions into code. Quite a large fraction of a programmer's working life consists of finding and correcting bugs.

Various weapons are available to us in the never-ending war against bugs. Perhaps the most basic is summarized in the slogan "Keep your interpreter happy!" – meaning, in the case of Perl, always run it using the `-w` warning option, and when warnings are generated, investigate them and modify your program to eliminate them, even if the program seems to be performing correctly. A warning is a sign that something is odd in the code; even if it does not lead to errors with the data you have fed into the program so far, it may well lead to errors with other data, or after other parts of the code are modified – intermittent bugs are the hardest ones to cure.[36]

Plenty of bugs will never be caught by `-w`, though.[37] If studying your code on paper does not reveal where it is going wrong, there are all kinds of *ad hoc* ways to get the machine to tell you.

I mentioned in chapter 6 the technique of inserting diagnostic `print` statements at key points in one's code, to check whether some variable has the value which it ought to have at that point. Or, if an error seems to be related to the fact that, say, a variable `$x` ought to be positive at a given point and you suspect that it might not be, you can try inserting a diagnostic such as:

```
if (not ($x > 0))
  {
  die "$x not positive at line XXX\n";
  }
```

Inserting diagnostics like these into one's code is a recognized debugging technique, and it can work fine, particularly if there are not too many bugs.

This technique can be cumbersome, though, specially if bugs prove numerous. Each diagnostic statement is extra code to be written (and, once its work is done, it must be deleted with care to avoid disturbing the "real" code surrounding it). Perl incorporates a *debugger* facility, which eliminates the need to insert special diagnostic lines into the middle of the code which is doing the actual work of your program, and which provides great flexibility for the task of tracking bugs down.

The debugger is invoked by running Perl with the `-d` option on the command line (i.e. `perl -d`). Rather than (as normal) running the program, too rapidly for a human being to follow, until it runs out of code or hits an `exit` statement or the like, in response to `-d` the system will initiate a dialogue with the user: the user will be able to control line-by-line progress through the program code, and can ask questions about the state of play at any point.

By far the easiest way to give the reader a feel for how the debugger works is to show an extended example. Let's run the countyCalcs program (33) of chapter 18 in debug mode.

Download free ebooks at bookboon.com

Figure 3 is a screenshot of a dialogue which a programmer might have with the debugger, initiated by the command

```
perl -d countyCalcs.pl
```

(the `%` symbol at the beginning of the first line of Figure 3 is the user's system prompt).

```
% perl -d countyCalcs.pl

Loading DB routines from perl5db.pl version 1.28
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(countyCalcs.pl:1):     open(INFILE, "<countyDataPlus.txt") or die
("can\'t open countyDataPlus.txt");
  DB<1> s
main::(countyCalcs.pl:2):     $i = 0;   #initialize index for rows of
@counties
  DB<1> p $i

  DB<2> s
main::(countyCalcs.pl:3):     while ($line = <INFILE>)
main::(countyCalcs.pl:4):       {
  DB<2> p $i
0
  DB<3> b 11
  DB<4> c
main::(countyCalcs.pl:11):      $name = despace($name);
  DB<4> p $name
Bedfordshire
  DB<5> c
main::(countyCalcs.pl:11):      $name = despace($name);
  DB<5> c
main::(countyCalcs.pl:11):      $name = despace($name);
  DB<5> c
main::(countyCalcs.pl:11):      $name = despace($name);
  DB<5> p $name
Cambridgeshire and  Isle of Ely
  DB<6> b 12
  DB<7> c
main::(countyCalcs.pl:12):      $counties[$i]{name} = $name;
  DB<7> p $name
Cambridgeshire_and_Isle_of_Ely
  DB<8> q
%
```

Figure 3

The three lines following that command line are standard announcements by the debugger which it gives whenever it is invoked, and after that it displays the first line of the countyCalcs program.

After showing this first statement, the debugger gives its prompt, `DB<1>` – the debugger prompts are always underlined, and the user's inputs are in bold.[38]

Because the user is feeling his way in an unfamiliar system, his first input is a cautious **`s`** meaning "move a single step forward". The debugger responds by showing program line 2, which initializes the variable `$i`. At this point the user finds that he is unsure whether the line of code displayed by the debugger at any point is the statement which has just been executed, or the one which is about to be executed: to discover the answer, he prints the current value of `$i` (**`p`** means "print"). The system responds with a blank line: the program line displayed is always the statement which is *about to be* executed, and before line 2 is executed Perl does not yet know about `$i`.

To check, the user takes one further step forward (at which point the debugger displays the next code lines:

```
while ($line = <INFILE>)
  {
```

– two lines which logically belong together as a unit), and again asks for the value of `$i` to be printed. This time it is shown as zero, because line 2 has been executed.

Please click the advert

SIMPLY CLEVER

ŠKODA

Do you like cars? Would you like to be a part of a successful brand?
We will appreciate and reward both your enthusiasm and talent.
Send us your CV. You will be surprised where it can take you.

Send us your CV on
www.employerforlife.com

Download free ebooks at bookboon.com

Now the user is ready to look at an aspect of the program where errors are more likely. Line 11 is the "despacing" line which is supposed to turn messy, multi-word county names into neat, all-black equivalents: there is plenty of room for programming errors there. The first few county names (Bedfordshire, Berkshire, Buckinghamshire) are not very challenging, but it would be interesting to see what happens to the next name.

Continuing to move through the `while` loop one step at a time with **`s`** would become excessively tedious, so instead the user inputs **`b 11`**, which makes line 11 (the line after `$name` has been assigned an "unreformed" string as its value) into a *breakpoint*. Then the user can input **`c`** ("continue"), meaning "run through the code until you reach a breakpoint". (The numbers within angle brackets in the debugger prompts rise with each user "action": printing a variable value, or setting a breakpoint, count as actions, but stepping or continuing do not count as actions by the user.)

After the first **`c`**, the user prints the value of `$name` and finds that, as expected, it is `Bedfordshire`. Three more **`c`** inputs cause the program to make further passes through the `while` loop, and then another **`p`** confirms that the last of these passes has reached the interesting county name. Now the user sets a second breakpoint at line 12, after `$name` should have been "despaced", and continues to that line. When **`p $name`** is entered again, the county name is shown in what turns out indeed to be its correct despaced form.

The user is now happy (for the time being, at least!), so enters **`q`** (quit) and returns to the system prompt.

On this occasion there was no bug to be found and corrected. But it is clear that when things do go wrong, the debugger offers a very powerful mechanism for seeking out the root of the error. The debugger commands we have looked at – step, set breakpoint, continue, print value – are only a handful (though probably the most useful handful) among dozens of different commands that the debugger recognizes. The command **`W`** sets a "watch" on a particular variable – rather than setting breakpoints at lines 11 and 12 and printing out the current values of `$name`, we could have begun with the command **`W $name`**, in which case the debugger would have reported on each occasion when `$name` was given a new value. We can override our program code by imposing a new value on a variable at a given point in code execution, and looking to see whether the consequences of that are what we expect them to be. And the debugger offers many other possibilities for monitoring the performance of our code and finding out precisely where it is going wrong.

Download free ebooks at bookboon.com

21 Beyond the introduction

At the beginning of this book and frequently thereafter, I have stressed that it is an introductory textbook for beginners, and cannot offer comprehensive coverage of the entire Perl language. This closing chapter aims to give a sense of what else is out there in Perl, waiting to be studied when you are ready.

Some of what is missing from the book is "more of the same". For instance, chapter 7 examined a few of the most useful Perl built-in functions; there are plenty more built-in functions that we have not mentioned. And often we have looked only at the most typical variant of a given Perl construction, neglecting permissible alternatives. In chapter 7 we saw that the built-in function `substr()` commonly takes three arguments, but it can take two, or four. Very many Perl constructions have variants which omit an element from the default structure, or include some extra element, with defined meanings in each case – normally we have looked only at the default structure.

Usually, the things we have left out do not enable the programmer to achieve anything that could not be achieved (perhaps less elegantly) using the constructions that have been introduced. Under the flow-of-control heading, for instance, alongside the familiar `while(condition){...}` loop there is an alternative which was not mentioned above, `until(condition){...}` – an `until` loop executes the block represented by the dots so long as the condition is *not* true. The construction `until(condition)` is precisely equivalent to `while(not(condition))`, but sometimes it can be easier to think about the condition in positive rather than negative terms. Again, the keyword `next`, partway through a loop body, enables a pass through the loop to be terminated prematurely on a specified condition, with control returning to the beginning of the loop without traversing later statements – but we can do the same by putting the later statements into an `if` block, though this may feel clumsy.

Where language elements omitted in this book do things which are truly additional to what can be achieved with the elements that have been covered, the extra things are often system-related matters which beginners would probably not want to get involved with. This applies to many of the built-in variables; for instance, `$^O` holds the name of the operating system for which the version of Perl you are using was built, and `$^M` is used to create an emergency memory buffer.

More serious is the fact that we have avoided the topic of *references*, which were mentioned in chapter 15. The consequence of omitting that topic has been that we have had to work with arrays and hashes with "one hand tied behind our back". Once one understands references and dereferencing in Perl, using multi-item data structures becomes a good deal more straightforward. Leaving references out was a teaching-strategy decision (which may have been the right choice for some readers and wrong for others). In my experience, the abstractness of the reference concept is confusing and offputting to many beginners, and leaves them thinking that programming is not for them, even though they could become successful programmers if they focused first on the more concrete, graspable aspects of Perl. (And again it is normally possible to achieve what one wants to do without using references.) When you are confident with the aspects of Perl covered in this book, I encourage you to go on and broaden your knowledge of the language, and references will be an early topic to look at.

Download free ebooks at bookboon.com

Apart from gaps like these, though, there are entire topics which have not been mentioned so far. Readers will want to be given an impression of what those topics are, even though we cannot go into details.

There has been no mention in previous chapters, for instance, of *object-oriented programming*. Successful large-scale software engineering depends heavily on techniques for implementing the divide-and-conquer principle discussed in chapter 16. Many readers will know that over the last twenty years or so, the object-oriented style of programming facilitated by languages like Java, C++, and C# has become a highly-regarded approach to software development: the abstract world controlled by a program is envisaged as containing objects of various classes, capable of interacting with one another in ways that depend on class membership.

Because Perl with its sophisticated pattern-matching facilities is so heavily used for text-processing applications, such as website management, object orientation arguably tends to be less central for Perl programmers than for those who work in other languages; text processing does not seem to lend itself well to the object-oriented style. But for those who want object-orientation, Perl supports it. It has machinery to define classes, construct new class instances, and so forth, using keywords which have not appeared in this book.

Another aspect of "divide and conquer" is the idea that a large-scale system of software will not all be one continuous program in a single electronic file, but will typically comprise separate, independently-developed programs held in separate files which are enabled to interact with one another via well-defined interfaces. In Perl these independent components of software suites are called *packages*. The package concept is clearly esssential when different parts of a large software project are implemented by different members of a team, but Perl packages are useful to solo programmers also. (For instance, object-oriented programming in Perl depends on its package mechanism.)

Many programming problems arise over and over again in numerous software projects, so that it is wasteful for separate individual software developers at different locations to keep reinventing the wheel. It is much more economical of expensive thinking time if a well-coded, robust solution to a problem can be made available for programmers everywhere to re-use. A large number of very general-purpose packages will be included (in files called *modules*) as a standard part of the Perl distribution already sitting on your local system; if you know about a standard module that would be useful, you can give your own code access to it via a `use` statement. But the worldwide Perl community encourage much wider re-use, of application-oriented as well as system-oriented code, by maintaining the website *CPAN*, the "Comprehensive Perl Archive Network" (www.cpan.org). CPAN offers Perl modules written by thousands of authors and relating to almost any application area you can think of, for free downloading and incorporation into your own project. (And CPAN supports Perl users in other ways too.)

Download free ebooks at bookboon.com

Finally, since Perl is so suitable for manipulating HTML files and the World Wide Web is so significant an element of life in the 21st century, a very important area of Perl is the set of mechanisms by which Perl code on a web server can communicate with client machines browsing the Web, and thus make a website interactive. To build a static website that simply displays pages to a visitor, all we need is HTML; but if we want the visitor to be able to fill in and submit an order form, or play a game, or do the many other things that involve two-way interaction between site and visitor, then our HTML will need to be mingled with program code supporting those interactions. Often, that code is written in Perl. Discussing how this works would take us far beyond the purview of the present book, but the topic has been a major reason in practice for the popularity of Perl as a programming language.

And here we shall end. A reader who has worked through this book should be well equipped to begin programming in Perl and to extend his or her mastery of the language as the need arises. God speed your code!

Please click the advert

With us you can shape the future. Every single day.

For more information go to:
www.eon-career.com

Your energy shapes the future.

e·on

Endnotes

[1] We all know nowadays that women can do the same jobs as men and are entitled to the same respect – it is not necessary to labour the point. From now on, rather than clumsily repeating "he or she" I shall follow the traditional convention that "he" covers either sex when referring to hypothetical or unknown people.
[2] Perl programs are sometimes called *scripts* – but this is really a distinction without a difference. The term "script" in a computing context has connotations of lightweight brevity, and it alludes to the way that significant work can often be done in just a few lines of Perl (whereas, in older languages, the shortest worthwhile programs might be much longer). But, short or not, a "script" is a program; in this book we shall use the latter word.
[3] I call this a "magic line" because, even if you need to use it and it works, when one is a beginner it is not worth trying to understand why it works.
[4] Languages of that kind are said to be *strongly typed*.
[5] Perl does not include specific symbols meaning "true" and "false". It does not need to, because in a line like `if ($a > 100)`, as soon as the material inside the brackets is calculated to be true or false, that value is immediately used by the `if` construction. If you try to *force* Perl to display symbolic representations of truth-values, for instance by writing `print($a > 100)`, it will represent "true" as the number 1 and "false" by "printing the empty string" (i.e. by displaying nothing at all) – but this is not a sensible thing to make Perl do.
[6] The term *keyword* is used for symbols written alphabetically such as `if`, `print`, `eq` which have defined meanings within Perl.
[7] The keywords `and`, `or`, `not` have non-alphabetic near-equivalents `&&`, `||`, `!`. There are differences in Perl between the former and the latter set of symbols, but the differences are too specialized to go into here.
[8] For the same reason one cannot use commas, e.g. `56,237`, either. But the underline character can be used to create visual grouping in the digits of a long number: `12_345_678` is treated by Perl as identical to `12345678`.
[9] Thomas Plum, *Learning to Program in C*, Prentice-Hall, 1983, p. 4-2.
[10] The phrase "built-in" refers to the fact that these are functions which Perl provides for you as part of the language. In chapter 16 we shall see that we can add further functions which we define for ourselves.
[11] More precisely, `<>` will read from `STDIN`, the "standard input", and the standard input is the keyboard unless the user changes it (a possibility which we shall not explore).
[12] It is characteristic of Perl to offer several alternative ways of doing a given thing, which provide no advantage (or at best allow a tiny saving of typing effort) at the cost of burdening the memory with extra constructions to learn. Perl enthusiasts often suggest that this is a positive virtue in the language, but that seems questionable. In this book I usually discuss the basic way to do anything, and pass over in silence various alternative ways to do exactly the same thing; but sometimes it seems necessary to mention an alternative, for a reason such as the one identified above.
[13] The Perl concept "word character" has more to do with computing than with words of the English language. Very few English words contain digits; on the other hand, plenty of words, such as *don't*, *o'clock*, contain apostrophes, but `'` is not a "word character".
[14] So, although we said above that pattern matching typically seeks to match a pattern *within* the target string, a pattern of the form `/^...$/` will be satisfied only if the material represented by dots matches the *entire* target string.
[15] As well as plus-sign for "one or more" and asterisk for "zero or more", Perl also has question-mark for "either zero or one". The pattern `/ab?c/` matches either `abc` or `ac`.
[16] Since it is quite easy to make the mistake of setting up an infinite loop, it is a good idea to find out at an early stage what the interrupt key-combination is on the system you are working at. But it varies from system to system, so unfortunately I cannot tell you the answer.
[17] For completeness I should mention that Perl has a further data type, the *typeglob*, with `*` as prefix symbol. However, this textbook will not discuss typeglobs.
[18] It is actually possible to change the default settings of Perl so that array elements are numbered from 1 rather than from 0. But the experts all advise against doing that.

Download free ebooks at bookboon.com

[19] The number of elements in `@words` will not change provided it does already have at least four elements, i.e. it has a slot 3, counting from zero; and if `@words` has the contents we gave it earlier, it has seven elements. On the other hand, if `@words` were an array of fewer than four elements, then assigning a value to `$words[3]` would cause it to be extended far enough to have a slot 3 to accept the value (any lower-numbered slot which did not previously exist would be given an empty placemarking value, pending assignment of some "real" value); and if `@words` did not previously exist at all, an assignment to `$words[3]` would cause it to be brought into being, with empty values in slots 0, 1, and 2.
[20] This depends on the target string having only single spaces between words. If it contained a sequence of two spaces, one element of the array returned by `split()` would be the length-zero empty string which occurred between the adjacent spaces. To treat sequences of whitespace characters as single splitting points, use the pattern `/\s+/` (as in the example below).
[21] The argument to `print` (and indeed to any function) is always in principle a list, though often a one-item list, so that a statement like `print $b` is really tolerated shorthand for `print($b)`. Perl is so easygoing that it often allows brackets to be omitted even round multi-item lists; instead of the statement in the text above, it would have worked equally well to omit the brackets round the list of items to be printed:

```
print $countyNames[12], "\t", $countyPops[12], "\n";
```

But, for the newcomer, the priority is understanding what is going on in the language, not learning all the shortcuts which permit a few keystrokes to be saved. I shall be consistent about including brackets round multi-item arguments to `print` (though since I have already been writing single-item arguments to `print` without brackets, I shall continue doing that).
[22] Steve McConnell, *Code Complete: a practical handbook of software construction*, Microsoft Press, 1993, p. 236.
[23] Since an array in a scalar context gives the size of the array, you might expect that a *list* in a scalar context would give the length of the list. But it doesn't:

```
$a = (15, 20, 25);
print "$a\n";
```

```
25
```

A list in a scalar context gives its last member. This is a good example of the unpredictability of non-scalar to scalar conversion, and justifies my recommendation in chapter 13 to avoid being imaginative in how one uses lists. If I wanted to take the last element of a list, I would turn it into an array and use `pop()` or the `[]` notation.
[24] This is actually a severe oversimplification, as I shall explain shortly. But for many purposes Perl works *as if* arrays could themselves contain arrays.
[25] For the benefit of readers who have some familiarity with C, references are in fact the Perl equivalent of what C calls pointers.
[26] This is not true for many other programming languages – a point we shall discuss below.
[27] In this case, the brackets are compulsory. In chapter 13 we saw that brackets round a list of function arguments can often be omitted. To my mind it is much easier always to put brackets round lists than to remember which brackets are optional and which compulsory.
[28] In the more "ordinary" programming languages which incorporate the argument names into the line introducing the function name, a user-defined function is commonly required to take a fixed number of arguments.
[29] Function (22) is a case where error-trapping that would be needed in practice has been omitted in this textbook for clarity. Suppose that `aveWordLen()` were called on some occasion with a set of arguments *all* of which, like *99p*, contained digits and hence did not "count" as words: then in line 14 the function would attempt to divide zero by zero, which would cause the program to crash with an uninformative error message. A robust program would ensure this never happened, by inserting before 22.14 a line such as `if ($N_words < 1)` followed by a block defining some suitable response to this special situation. The nature of the response might depend on details of the

Download free ebooks at bookboon.com

wider program which uses `aveWordLen()`: perhaps it would be most convenient for the program to "die" with an error message tailor-made to the situation, or perhaps the function should return some special value such as 0 or –1, but either way line 22.14 would never be reached.

[30] In real life, one would probably use a "stop list" to avoid counting common, uninteresting words like *the* or *is*. To keep things simple, we will leave out that step and count every word.

[31] The particular "hashing algorithm" suggested here (multiply ASCII codes together and take the remainder modulo 1000) is too simple to work well in practice; but it should serve to illustrate the essential nature of hash tables (and to explain why there is no obvious, visible logic to the sequence in which items are stored in them).

[32] Readers might wonder, if we are going to end up working with exact figures for the county data, why in chapter 10 we used populations rounded to the nearest thousand. That was done purely in order to reduce the clumsiness of code snippet (9). In line 9.4 it was awkward to have to repeat the sequence `[0123456789]` four times; if we had been using exact figures, it would have had to be repeated seven times! Of course, by the end of chapter 10 we knew that that pattern could be represented compactly as `\d{7}` – but when we reached (9), this construction had not yet been introduced.

[33] Since some of the county names contain internal spaces, in practice it would be odd for a data file to use nothing more distinctive than space to separate the fields of multi-field lines. If that did happen, we could use the contrast between letters and digits to work out where names ended and numerical data began – but using a black character such as the vertical bar as field delimiter is more realistic.

[34] Some readers may be familiar with `printf` in the C language, and they can skip the following section; `printf` in Perl is more or less identical.

[35] For those readers who are already familiar with C there is a pitfall here. C contains a keyword `argv` which works very similarly to Perl's `@ARGV`, but, in C, `argv[0]` denotes the program name, and the first argument to the program is `argv[1]`. In Perl, the array `@ARGV` includes only the arguments on the command line, not the program name immediately preceding them, so if there is just one argument it will be `$ARGV[0]`.

[36] The original slogan, coined by a wise programmer whose identity I have unfortunately forgotten, was "Keep your compiler happy!" It happens that Perl is an interpreted rather than a compiled language, but the principle is the same (and if you do not know the difference between these two kinds of programming language, in the present context that really does not matter).

[37] Apart from `-w`, it is possible to make Perl even more resistant to questionable code via the "pragma" `use strict`. But discussing that would be beyond the purview of this introductory textbook, and anyway even `use strict` can only detect a small fraction of potential bugs.

[38] Elsewhere in this book, I have used roman versus italic to differentiate program code from output generated by a program, but that was just a convention adopted in the book to make things clearer to the reader. In debugger dialogues, underlining and bold face really do appear on screen as shown in Figure 3.