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.

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.)

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!

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.

[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

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.