



CHAPTER 1

Introduction

- 1.1 On programming 4
- 1.2 On Perl 7
- 1.3 A bigger picture 15

To write a story, fiction or fact, one must grasp the basics of the language in use—enough at least so that one can make simple statements that may be understood by readers. One must also be able to string together a series of such statements in a manner that communicates meanings and events in relation to time and space. Stories also have certain compositional elements like a beginning, a middle, and an end, although the presentation need not proceed in that order.

Listen to writers talk about writing, and you'll find that writing is seldom the result of any "holistic inspiration" where the writer realizes the entire story at once in the mind and simply writes it down. Much more often, you'll find writers claiming that inspiration is rare and fleeting and usually comes during—and as a result of—the writing process, not the other way around.

Programming is similar in many ways to writing. One must know the basics of the language and how to string together a series of statements such that events and meaning are described in space and time. Structural elements—beginnings, middles, and ends—are also important in program composition. Finally, real inspiration generally occurs while immersed in the development process, not before.

Whether programming is an art or a science is often a subject of discussion. But programming is neither an art achievable only by innately talented right-brain visionaries, nor a strictly scientific left-brain logic of first principles. It is a craft, like writing, that can be learned, practiced, developed, and honed to a variety of skill levels and for a multitude of purposes.

Some writers write great literary novels, others write pulp fiction, and many people who are not writers by trade write all kinds of narratives from business reports to postcards to notes on little squares of adhesively backed yellow paper. Similarly, some programmers write elegant programs solving massively complex problems; others produce solid utilitarian code daily; and many, who are not programmers by trade, crank out all manner of tools and big and little programs and go on with their real jobs.

So, if programming is a craft like writing, and writing is about telling narratives, what is programming about? Programming is about solving problems.

1.1 On programming

Computers are mindless devices capable only of doing what they are told. Before we talk about the activities of programming we need to have a basic understanding of what a program is and the role it plays in turning an expensive piece of mindless hardware into a useful device.

Imagine the following scenario: You are taken to a room containing a desk and a chair. On the left of the desk is an "in-box" full of pages of numbers, and on the

right is an empty “out-box.” In between lies a manila envelope, a calculator, a pad of paper, and a stack of blank forms. You are told to open the envelope and follow the instructions you’ll find inside. The simple and explicit instructions tell you to take the first page of numbers from the in-tray and perform certain simple arithmetic operations on particular sets of those numbers and to write the results of these calculations into certain numbered boxes on one of the blank forms. When you’ve completed one page of numbers, you must place the newly filled out form into the out-tray and begin again with the next page of numbers in the in-tray.

In this scenario, you are operating essentially as a mindless computing device: taking *input*, performing simple operations, and writing out the results. The instructions are simple and do not require “thought” or interpretation, merely that you can read in numbers, operate the calculator, temporarily store intermediate results on the pad of paper, and control the *output* device (pencil) to write out the results (see figure 1.1).

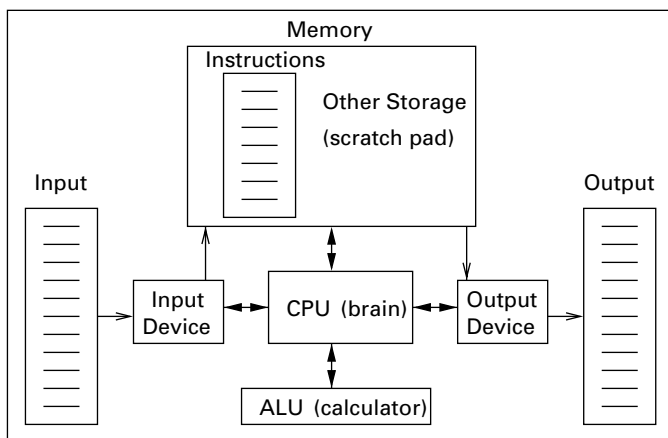


Figure 1.1 Processing data

An important point to note is that you have no idea about what you are doing in this scenario. You may be just one part of a team performing various steps in a complex encryption/decryption scheme, or you might simply be balancing my checkbook. At any rate, you would not likely enjoy this particular activity. In addition to being boring and repetitive—the usual definition of mindless—such activities nonetheless require attention to many small details. They occupy your brain, but don’t absolutely free your mind. It would be hard to meditate while performing such a task.

The most important component in the above situation is the set of instructions. The mindless brain can be replaced by a mindless *central processing unit*

(CPU) controlling simple mechanical/electrical devices for input, output, arithmetic operations and temporary storage. However, it takes a mind to conceive a given set of instructions to control such a machine. Anyone who has stayed up late on Christmas Eve trying to put together a “some assembly required” toy for their child can appreciate the problems of working with incomplete, out of order, and badly written instructions.

I said programming is about solving problems, but that is not entirely accurate. You not only have to think of a way to solve a given problem, you also have to develop that solution into a complete step-by-step set of simple instructions for solving the problem. When a method for solving a problem is reduced to a series of simple, repeatable instructions, we call that set of instructions an *algorithm*.

An algorithm can be specified in any language, but regardless of how complex the algorithm might be, each step or instruction must be simple and not subject to interpretation or intuition. An algorithm must also always generate the same result. If each step is performed flawlessly, the outcome is guaranteed. Programming is about creating algorithms to perform simple or complex tasks and translating these algorithms into instructions that a computer can perform.

The particular instructions that a computer can execute are in a form known as “machine language” or simply “machine code.” Machine code instructions consist of sequences of numbers, ultimately reduceable to zeros and ones (i.e., binary code). These instructions deal with low level operations such as storing a number into a particular memory location, reading a number out of a particular memory address, or adding a number into an accumulator. For example, the following snippet of machine code causes two numbers, which are stored in memory, to be added together and saved into another memory location for later use:

```
8B45FC 0345F8 8945F4
```

Of course, we humans have a hard time trying to formulate algorithms in a machine language so we soon developed a language consisting of abbreviated words to stand for the particular operations a given machine could perform. We could write programs out in this “assembly” language, then translate the language into the corresponding machine code using one machine instruction per each assembly instruction.

Assembly code was easier to write than the machine code it replaced, but because different machine architectures used different machine code, each had to be programmed in its own version of an assembly language. A program written for one machine type had to be rewritten to run on another machine type. Another problem with assembly languages is that they still dealt with the low level instructions of moving around bits of information.

Higher level languages were later developed to allow instructions to express higher level concepts. With a low level language like an assembly, one had to write individual instructions to place a number from a specific location in memory into a temporary holding area; add a number from a different memory location; and, finally, store the result of that operation in a third location in memory. With a high level language, one could write a single instruction that encapsulates the concept (see figure 1.2).

Machine Code	Assembler Code	High Level Code
8B45FC	<code>movl a, %eax</code>	
0345F8	<code>addl b, %eax</code>	<code>c = a + b</code>
8945F4	<code>movl %eax, c</code>	

Figure 1.2 Comparison of machine, assembly, and high level languages

Aside from the obvious advantage of making the instructions easier to read and write for humans, compilers were created that translated each high level statement down to the corresponding assembly and machine codes for different machines. This meant that one could write a program once in the high level language and be able to run it on any machine that had a compiler for that language.

Early high level languages included Fortran, designed primarily for mathematical computing; COBOL, for business related programming; and BASIC and Pascal, intended initially as teaching languages. Today, a wealth of high level languages exist from which to choose, each with particular strengths and weaknesses. The remainder of this chapter is about the language that will be taught in this book.

1.2 *On Perl*

Perl was created by Larry Wall and initially released in 1987. Perl's inception is illustrative of its nature (and certainly says something about its creator as well). When Larry was faced with a problem that involved complex text processing and report generation exceeding the capabilities of what could be done easily with standard Unix tools like `sed` and `awk` he made a choice. Rather than restricting his viewpoint to solving this particular task, he saw that his problem was just one of a whole class of problems for which existing tools and languages provided no simple solutions. So, in the spirit of long-term laziness, Larry Wall created a new tool for solving such problems.

Perl did not simply fill a gap in the existing toolset. Perl incorporated the capabilities of existing tools and borrowed freely from other languages. It soon

became the language of choice for getting things done in the Unix environment (and its usefulness soon spread to other operating systems and environments). Put another way, Perl didn't fill one particular gap in the toolset; it became *the* tool to use for filling gaps everywhere.

Designed not only for practical usefulness, but also for continued expansion, Perl has become a powerful general purpose programming language. In its current incarnation, it offers extremely powerful regular expression enhancements, object-oriented programming, a well defined module system, references, nested data structures, and quite a bit more.

Perhaps some, or even all, of the features I just mentioned don't mean a lot to you at the moment. You want to know what *you* can do with Perl. Telling you that you can write programs to do anything you want wouldn't be accurate, but it wouldn't be too far from the truth either. Perl isn't well suited for some kinds of programming: for example, writing operating systems, writing device drivers, and doing heavy numeric analysis. So what is Perl good for?

Perl excels at reading, processing, transforming, and writing plain text. Processing text may not seem like a big deal until you realize all the ways you might want a program to interact with textual data. Not only the plain text files you save on your hard drive, but the file and directory names themselves are representations of textual data. With Perl, you can easily write code that will read in directory listings, create new files or directories, rename files, delete (unlink) files, and more. Automating many kinds of system administration and backup tasks is but one common use of Perl.

Client-server interactions, such as the Structured Query Language (SQL) statements you send to a database server and the results it returns, or the HTTP requests you send to a webserver and the HTML pages it sends back, are all largely plain or simply encoded text. Similarly, communications between a webserver and external programs use a simple, encoded text protocol known as Common Gateway Interface (CGI). A significant portion of Internet sites that provide dynamic content or search capabilities are powered by Perl programs that handle requests from the webserver; turn these into queries for a database's server; collect, transform, and format the resulting data; and pass it back to the webserver. Because Perl makes it so easy to write programs that work between other programs, Perl is often referred to as a "glue" language. And because of its wide use sticking various things together throughout the Internet, Perl has also been called the "duct tape" of the Internet.

I mentioned that Perl was designed to grow and that it has a well defined module system. This means that many of the common tasks—and some uncommon ones—have already been encapsulated into *modules* that are freely available for anyone to use (see the discussion of CPAN later in this chapter). When you

want to write a CGI program, write client software to do file transfer protocol (FTP), or interact with a database, you can use a well tested and *debugged* module that provides most of what you need, leaving you only to add the code to deal with your specific task.

Not only do modules make it easy to use Perl for common tasks, but some modules provide extensions that allow you to use Perl for problems for which Perl itself might be less well suited. I mentioned above that Perl may not be a good choice for numeric analysis. However, there are modules designed to extend Perl's handling of numeric data by providing arbitrarily large integers and floating point numbers. There is also the Perl Data Language (PDL) module package, which provides extensions for doing fast mathematical computations on large matrices of numeric data.

Some programming languages are *interpreted*, meaning that a program written in that language is read by an interpreter program that translates each statement into the appropriate machine code and executes it. In such languages, errors cannot be detected until the program is already running and the interpreter encounters a statement that generates an error. Other languages are *compiled*, meaning that a whole program is read by a compiler and translated into machine code before it can be run. In this case, many kinds of errors can be caught before the program is even run. (Some errors, called *run-time errors*, will still not be caught until the program is running.) Perl is both interpreted and compiled. When you run a Perl program, Perl reads the entire program and compiles it into an internal format (not machine code), then interprets this internal representation like a regular interpreter. Thus, in many places in this book, I will sometimes refer to the perl interpreter and sometimes to the perl compiler—but they both refer to perl itself. Perl, however, refers to the Perl language.

The advantage of such a compiled/interpreted system is that many kinds of errors can be detected during the compilation phase, before a program begins executing. Yet you do not need to separately compile and link your code into a binary executable each time you make a change, as you would for a strictly compiled language such as C. You can simply type in your program and run it. The perl compiler/interpreter takes care of the rest. The tradeoff is a little loss in speed. A program compiled into machine executable code runs somewhat faster than one whose statements are individually interpreted with each run through. That said, Perl programs usually run very fast, especially for text processing types of tasks.

Yet another benefit of Perl's interpreted nature is memory management. In many lower level compiled languages, like C, your program is required to deal specifically with allocating and releasing the necessary memory for storing data while your program is running. When you program in Perl, the perl interpreter takes

care of allocating extra memory when needed and releasing that memory so it can be used again when it is no longer needed. This doesn't mean you can completely ignore memory issues. You still need to make choices such as whether to read a file completely into memory or read a file one line at a time. But you don't have to worry about actually allocating and releasing the memory yourself.

Later in this chapter, I will continue this discussion of Perl and why it is the greatest thing since peanut butter (sliced bread really wasn't such a big deal until peanut butter arrived on the scene). Right now, let's turn to the more practical matter of ensuring that you have a perl distribution up and running so that you can begin your Perl programming journey.

1.2.1 Getting started

If you are not using a system that has perl installed, you will have to obtain a perl distribution and install it yourself. The latest source distribution can be found on the Comprehensive Perl Archive Network (CPAN) at <http://www.perl.com/CPAN/src>. Pointers to binary distributions for various platforms can also be found there in the */ports* directory. The distributions contain detailed installation information, and the process is usually painless.

Unix-like systems On a Unix-like system, you will probably want to compile your own version of perl, assuming you have a C compiler installed. The process is simple, though it can be a little time consuming. The first thing you want to do, after downloading the latest distribution from the above mentioned CPAN site, is to unpack it, go into the resulting directory, and read the *README* file for your system and the *INSTALL* file. This should provide you with enough information to build your own version of perl. Essentially, the process is just

```
$ rm -f config.sh Policy.sh
$ sh Configure
$ make
$ make test
$ make install
```

The configure step will take awhile. You will have to answer a variety of questions about your system and where you want things installed. Picking the defaults is usually all that's needed in most cases.

Win/NT systems On Win32 systems, your best bet is probably to obtain the ActiveState version of perl, which is available at <http://www.ActiveState.com/>

For Win95, you will probably need to get the DCOM package and install it before starting to install the perl distribution. You can find it at http://www.microsoft.com/com/dcom/dcom1_2/download.asp

Installing the ActiveState version of perl is a matter of double-clicking the archive. The install process will ask a few questions and you should accept the defaults unless you have good reason not to accept them. After this, you should be able to run perl from the command prompt and run the `perldoc` utility to access the documentation (see later in this chapter).

MacOS You can get a compiled binary distribution of Perl for the MacOS in the ports section of CPAN: <http://www.perl.com/CPAN/ports/index.html#mac>. This link should automatically redirect you to a nearby mirror site.

Installing this version involves unpacking the archive, starting the program, and setting a few configuration details that are pointed out in the included *README* file. The major sections of the documentation should be accessible via the help menu.

1.2.2 *Running Perl*

Once you have perl installed, creating and running a Perl program is a simple process. The following is a simple, one line program that prints the string “Hello World”:

```
print "Hello World\n";
```

Create a new text file (plain text) using any editor with which you are comfortable. This should be a text editor, not a word processing program—you can find a list of decent editors for various platforms by pointing your browser at <http://reference.perl.com/query.cgi?editors>

Now enter the above statement. Save the file as “first.” You can then run the program from the command line as

```
perl first
```

On many Unix-like systems you can create your script to be run as if it is an executable program. This method means adding a special first line to your program and setting the executable bit on the file with the `chmod` Unix command. Here is the new program:

```
#!/usr/bin/perl
print "Hello World\n";
```

The first line, called the “pound-bang” or *shebang* line starts with the two characters `#!` followed by the full path to where perl is located on your system—in other words, the absolute directory path to the perl program. If you save this as

before and then type `chmod +x first` at the prompt, you can then invoke the program like:

```
first
```

If the current directory is not in your `PATH` (the environment variable listing search paths for executable programs), you may need to qualify the above call as

```
./first.
```

The ActiveState port comes with a `pl2bat` utility to turn your perl program into a batch file that can be placed in your `PATH` and called like any other program.

If you are using MacPerl then you should be able to simply choose `new` from the `file` menu, type in the script, and then choose `run script` from the `script` menu. The shebang line is not necessary on non-Unix systems, but it is always a good idea put one in, because perl will check it for command line switches such as `-w` (see chapter 2). With MacPerl, you can also save the script as a Mac-specific item called a “droplet,” which is a version you can execute by double-clicking its icon.

1.2.3 *Getting help*

Perl is a relatively easy language to learn, but it is not a small language. This book does not attempt to be a reference for the Perl language. If you have perl installed, however, you already have the most up-to-date language reference available. The perl distribution includes a large amount of documentation that is installed as Unix manpages and/or HTML pages (or some other format, depending on installation configuration details). The raw documentation is in a plain text mark-up format called Plain Old Documentation (*POD*), and is also readable using the included `perldoc` utility (or `shuck` on the Mac). To view the initial perl pod-page you can enter `perldoc perl` at your command line prompt. This document provides a list of the remaining sections of the core documentation. Another useful starting page is `perldoc perltoc` which provides a more in-depth table of contents of the Perl documentation.

One extremely useful set of documents is the set of *perlfaq* documents. Very often one begins to tackle a problem by breaking it down into smaller problems and addressing those. However, when learning a new language, some of the smaller problems are often difficult because you do not yet know how to express the solution in the context of the language you are learning. This is when it is time to turn to the Frequently Asked Question (FAQs).

Do not assume that the FAQs only address simple or “little” questions and that your question will not be found there. Many FAQs do have simple answers, but they are no less valuable for that. On the other hand, there are also many real

programming issues addressed in Perl's FAQs. No matter how easy or difficult your particular problem seems to be, you'll often have good luck finding something of use in the FAQs. In chapter 3, we will begin developing a tool to quickly search the FAQs for information we might need.

The FAQs are divided into nine sections, or files, named *perlfqa1* to *perlfqa9* and are viewable with the `perldoc` utility. The *perltoc* page describes each of these and lists all of the questions you will find answers to in each document.

Another source of help is the Usenet community. There are separate newsgroups for discussions on miscellaneous Perl topics (*comp.lang.perl.misc*), discussions on perl modules (*comp.lang.perl.modules*), and the Tk graphics toolkit (*comp.lang.perl.tk*). There is also a moderated group that you can read, but participation requires registration (*comp.lang.perl.moderated*). If you have never participated in Usenet newsgroups before, I recommend that you first take a look at *news.announce.newusers*.

Although all of the newsgroups are open to public participation, they are not forums for questions addressed in the perl documentation and FAQs. The people participating in these groups are knowledgeable and helpful, but you are expected to have tried to find answers to your questions in the documentation before turning to the newsgroup. These newsgroups are not free help desks. If you treat them as such, you will likely get ignored or worse.

On a related note, remember, programming is about problem solving. Beginners often approach a programming language as simply another application to learn. This can lead to asking questions like "How do I do "X" in Perl?" When learning a new word processor application, one might formulate a question such as "How do I create footnotes in my documents?" But a programming language is not simply an application. It provides ways to formulate solutions to problems. It does not provide single commands or functions for every conceivable problem.

Before you ask a "How do I..." question, search through the documentation on Perl's built-in functions (see `perldoc perlfunc`) to see if one meets your needs. Then search the FAQs to see if your question has already been answered. If these two approaches fail, ask yourself how you would go about solving the problem without a computer. For example, recently a question appeared on the *comp.lang.perl.misc* newsgroup asking (and not for the first time) how to tell if an integer is odd or even. There is no simple `even` or `odd` built-in Perl function that solves this problem for you directly. The obvious question to ask yourself is "How do I tell if an integer is even or odd?" Most people simply notice if the last digit is one of 0, 2, 4, 6, or 8. If so, the integer is even. Algorithms often arise from such simple beginnings.

You may not succeed in solving your particular problem, or your solution may not be the optimal solution, but this common sense approach is the first step in

thinking like a programmer. One more avenue, before resorting to asking your question in the newsgroup, is to use one of the Usenet search engines (for example, *www.dejanews.com*) to search the Perl newsgroups for similar questions that may have been asked and answered in the past. Finally, if you have exhausted all avenues of inquiry without success, ask your question on the appropriate newsgroup. Be sure to include information on what you've tried so the helpful people there know you are not just looking for free handouts but are actually interested in learning.

Another place to begin exploring a wealth of Perl related information is the Perl home page at *www.perl.com*. From there you will find links to HTML versions of the Perl documentation and FAQs, plus pointers to the Comprehensive Perl Archive Network (CPAN). At CPAN, you can find and download not only the Perl source distribution, with its standard libraries and modules, but also a large number of contributed modules and scripts for various problem domains.

Jargon If you are new to programming and/or Usenet in general, you may encounter quite a bit of jargon when using the above-mentioned resources. One very good, and quite extensive, resource you may want to look at is the *Jargon File*. This is a large dictionary or lexicon of common and not-so-common slang terms found in the hacker community. You can find this file at <http://www.tuxedo.org/~esr/jargon/>, or use your favorite search engine to locate a copy.

Just to get you started, I'll run through a few terms that you might happen upon as you begin to read the literature. For example, if you do ask a question on the newsgroup and it is answered in the FAQ, you're unlikely to get responses beyond the standard RTFM which means, "Read The F***** Manual."

Quite often, people will quote material from the "camel" or "llama" books. These are two standard Perl books by published by O'Reilly and Associates. The books feature pictures of the respective animals on their covers and are titled *Programming Perl* (camel) and *Learning Perl* (llama)—See appendix C. Both are very good books, by the way.

A couple of other frequently used terms are "grep," meaning to search, and "grok," meaning to understand. Grep derives from the standard Unix search utility of the same name. If you want to grok Year 2000 (Y2K) issues in Perl, you should grep the FAQs for the relevant entries.

Another common term is "parse." Specifically, parse means to break up a piece of data—such as a string (a sequence of characters or words)—according to a specified set of syntactic rules. More generally, parse is often used in the context of simply recognizing and/or extracting particular bits of data from a larger chunk of data.

You may also see reference to "p5p," which refers to the perl5-porters, a group of people responsible for maintaining and upgrading the actual Perl distribution

across the many platforms on which it runs. Another group—set of groups really—is the Perl Mongers, a collection of user groups distributed around the world. (Visit <http://www.pm.org> to find your nearest group or to start one.) I happen to be a member of the Winnipeg Perl Mongers (Winnipeg.pm for short).

A variation of the Monger moniker seems to be Perl M(o|u)nger, which is regular expression talk (you'll learn about regular expressions in chapters 6 and 10) to refer to either Monger or Munger. Munger, then, is the noun form of the verb “to munge,” which means essentially, in the context of data processing, to parse, process, slice, dice, julienne, massage, fold, bend, or otherwise mutilate (i.e., manipulate) data.

This little interlude barely scratches the surface of the jargon you will run into in your journey. If nothing else, at least I've warned you that you are entering not only a new area of study, but a new linguistic arena as well. Don't forget to check out the *Jargon File* mentioned above—it contains much more than mere definitions.

1.3 *A bigger picture*

The previous sections have dealt with basic information on programming and on Perl. Now we will take a brief step back and take in a larger view. My first choice for a title for this section was “Practical and Philosophical Remarks on Programming, Perl, and the Rest of this Book,” but that was a tad long winded—even for me.

Several years ago, I worked as an inshore diver, doing a variety of underwater inspection, repair, and construction tasks, often in fast-moving water. Every job was different, and there was no such thing as a transportable stable work platform that could be used in every situation. Instead, we had the next best thing: a shop with a selection of tools, a wide variety of surplus construction steel (rings, bolts, rods, angle iron, and I-beams), and an arc-welder. This was a hacker workshop. We created reusable bracing and clamping components, and rigged up a variety of different scaffold systems that could be lowered from the barge, positioned, and anchored in various ways to bridge piers of different sizes and shapes, pilings, or dam gate systems. That arc-welder was the key to being able to quickly and solidly connect a variety of components into working solutions we could deploy in the field.

In the programming world, Perl reminds me very much of that workshop and, in particular, the arc-welder. Perl has been nicknamed the Swiss Army chain-saw of programming languages due to its multitude of built-in tools and overall brute force utility. I think a better analogy is that of a Swiss Army arc-welder, one quite capable of hacking out fast, sometimes crude, one-time solutions, as well as building and joining (virtually seamlessly) components for solving more complex or longer term problems.

Perl is not your average everyday programming language. In fact, it is an exceptional everyday programming language. There are currently more high level programming languages out there than you can shake a stick at. Of course, some computer scientists seem to get great pleasure from shaking a lot of sticks anyway, and Perl seems to have more than its fair share of detractors from computer science purists, who complain that Perl is too big, ugly, and redundant.

So what makes Perl so great? No single feature of Perl makes it outstanding, and any program you write using Perl could also be written using another language. Certainly, Perl makes some things easier to accomplish than they might be in other languages, but this can't be all there is to it. There are other languages that offer pretty much the same high level functionality as Perl, and do so in a way that seems to satisfy the above mentioned stick wavers. Yet Perl remains wildly popular. The language continues to evolve, and its user base is still growing unchecked. What appeal might Perl have beyond the sheer functionality that might first attract programmers from other languages, and why would experienced programmers fall in love with this language if other "better" programming languages exist?

I lay the blame squarely on the shoulders of Perl's creator, Larry Wall. (Well, more precisely, slightly above and between Larry's shoulders.) Besides being a computer programmer, Larry also has a background in linguistics. Consequently, the fact that Perl has the qualities of a natural language is no accident. You can read some of Larry's own musings on these qualities at <http://kiev.wall.org/~larry/natural.html>. Here I will touch upon only a couple of points in this regard.

Two of the things Perl receives criticism for are the richness and the redundancy in the language, which are by no means unrelated issues in Perl. By richness, I mean that Perl is a big language, incorporating and supporting a large number of powerful and specialized features directly in the language. In contrast, other languages tend to be minimal, providing standard libraries for specialized tasks. The list of Perl's specialized features includes regular expressions, pattern match operators, process management, file and directory manipulation, and socket programming tools, to highlight just a few. Another example of Perl's richness is its ability to provide more than one way of saying or accomplishing the same task. This is redundancy; it exists in natural languages and in Perl. Indeed, the Perl slogan is TMTOWTDI (pronounced "timtoady"), which stands for "There's More Than One Way To Do It."

Critics say that both richness and redundancy make the language harder to learn. But this is only true up to a point. Certainly, I may be able learn a complete minimalist programming language rapidly, but assuming there are libraries providing additional functionality, I would then still have to learn to use the various libraries to do the things I might want to do. Perl is not really more demanding in

this respect. You do not have to learn the entire language before you start, merely the essential elements plus any extra built-in features you find necessary for your present task. Similarly Perl's redundancy doesn't require that you learn every possible way to say something before you begin. Redundancy merely expands your options. Indeed, the positive consequence of these natural language qualities is not that they make it easier to think in Perl; they make it easier to express your thoughts in Perl.

There is also a strong sense of community among many Perl programmers. More than just a large number of users sharing information on various forums, the Perl community shares a sort of Perl spirit. Perl's "naturalness" lends itself to playfulness—not merely clever programming tricks, but poetry, puns, and other games of the sort people play with natural languages. Of course, this community spirit is more than fun and games, but I think it is significant to note that programmers coming from other languages who have found their calling to be growing tedious have expressed gratitude that Perl has made programming fun again.

The Perl community also holds a strong concept of sharing. Help and advice are given freely on the newsgroups and many talented programmers cooperate on the development and evolution of Perl itself. Beyond that is CPAN—a vast collection of contributed modules from Perl programmers all over the world. Programming, unlike most creative acts, places a high premium on reuse rather than originality. The Perl community is no exception. Several hundred modules are available on CPAN, ranging from database interfaces to development tools to interfaces for several graphics libraries to Internet programming to date manipulation modules and much more. Whenever you find yourself facing a new programming challenge, check out CPAN. Chances are pretty good that someone has written a module that will make your task much easier.

By now you might be wondering when we will stop talking about Perl and start learning to program with it. The next two chapters concentrate on aspects of writing good code and the process of developing programs. In the latter chapter, we follow the development of two Perl programs from initial idea through to working programs. A good deal of Perl will be presented along the way.