Perl 5 Tutorial

First Edition

(Release Candidate 3)

Chan Bernard Ki Hong

Perl 5 Tutorial
First Edition
Author: Chan Bernard Ki Hong (webmaster@cbkihong.com)
Web site: http://www.cbkihong.com
Date of Printing: September 26, 2003
Total Number of Pages: 245
Prepared from LaTeX source files by the author.

Important: Please note that this is a preview edition of the document and is released for collection of feedback purposes only. Therefore, drastic modifications may be made to this document at any time until it is completed and finalized.

In order to further enhance the quality of this publication, the author would like to hear from you, the fellow readers. Comments or suggestions on this publication are very much appreciated. Please feel free to forward me your comments through the email feedback form or the feedback forum on the author's Web site.

Contents

Preface

If you are looking for a free Perl tutorial that is packed with everything you need to know to get started on Perl programming, look no further. Presenting before you is probably the most comprehensive Perl tutorial on the Web, the product of two years of diligence seeking reference from related books and Web sites.

Perl is a programming language that is offered at no cost. So wouldn't it be nice if you can also learn it at no cost? Packed with some background knowledge of programming in C++ and Visual Basic, when I started learning Perl several years ago, I couldn't even find one good online tutorial that covered at least the basics of the Perl language and was free. Most Perl tutorials I could find merely covered the very basic topics such as scalar/list assignment, operators and some flow control structures etc. On the other hand, although I have accumulated certain levels of experience in a number of programming languages, the official Perl manual pages are quite technical with whole pages of jargons that I was not very familiar with. As a result, the book ***"Learning Perl"*** written by Larry Wall, the inventor of the Perl language, naturally became the only Perl textbook available. The O'Reilly Perl Series present the most authoritative and well-written resources on the subject written by the core developers of Perl. While you are strongly recommended to grab one copy of each if you have the money, they are not so cheap, though, and that's the motive behind my writing of this tutorial — so that more people with no programming background can start learning this stupendous and powerful language in a more cost-effective way.

Although this tutorial covers a rather wide range of topics, similar to what you can find in some other Perl guidebooks, you are strongly encouraged to read those books too, since their paedagogies of teaching may suit you more.

Here are several features of this tutorial:

- ⋆ As this is not a printed book, I will constantly add new materials to this tutorial as needed, thus enriching the content of this tutorial. Moreover, in order to help me improve the quality of this tutorial, it is crucial for you to forward me your comments and suggestions so that I can make further improvements to it.
- ⋆ In response to requests made from several visitors, this tutorial, in PDF format, has been made available for download. I hope this will help those who are charged on time basis for connecting to the Internet. This tutorial is typeset in LaTeX, a renowned document typesetting system that has been widely used in the academic community on Unix-compatible systems (although it is available on nearly any operating systems you can think of). In general, the PDF version will be updated prior to the online HTML version.
- ⋆ You will find a list of Web links and references to book chapters after each chapter where applicable which contains additional materials that ambitious learners will find helpful to further your understanding of the subject.

- ⋆ Throughout the text there would be many examples. In this tutorial, you will find two types of examples — **examples** and **illustrations**. Illustrations are intended to demonstrate a particular concept just mentioned, and are shorter in general. You will find them embedded inline throughout the tutorial. On the other hand, examples are more functional and resemble practical scripts, and are usually simplified versions of such. They usually demonstrate how different parts of a script can work together to realize the desired functionalities or consolidate some important concepts learned in a particular chapter.

- ⋆ If applicable, there will be some exercises in the form of concept consolidation questions as well as programming exercises at the end of each chapter to give readers chances to test how much they understand the materials learned from this tutorial and apply their knowledge through practice.

This is the first edition of the Perl 5 tutorial. It primarily focuses on fundamental Perl programming knowledge that any Perl programmer should be familiar with. I start with some basic ideas behind computer programming in general, and then move on to basic Perl programming with elementary topics such as operators and simple data structures. The chapter on scoping and subroutines is the gateway to subsequent, but more advanced topics such as references and object-oriented programming. The remaining chapters are rather assorted in topic, covering the use of filehandles, file I/O and regular expressions in detail. There is also a dedicated chapter on error handling which discusses facilities that you can use to locate logical errors and enhance program security. The final chapter on CGI programming builds on knowledge covered in all earlier chapters. Readers will learn how to write a Perl program that can be used for dynamic scripting on the World Wide Web. However short, the main text already embraces the most important fundamental subjects in the Perl programming language. In the appendices, instructions are given on acquiring and installing Perl modules, setting up a basic but functional CGI-enabled Web server for script testing, and there is a voluminous coverage of Unix fundamentals. As much of Perl is based on Unix concepts, I believe a brief understanding of this operating system is beneficial to Perl programmers. An appendix is also prepared to give my readers an idea of the internal structure of general hashes. While authoring of this tutorial cannot proceed indefinitely, topics that were planned but cannot be included in this edition subject to time constraints are deferred to the second edition. A list of these topics appear at the end of this document for your reference.

In the second release candidate of this tutorial I made an audacious attempt of adding into it two topics that are rarely discussed in most Perl literature. The first is the `Error` CPAN module for exception handling. The second attempt, which is an even more audacious one, is an introduction of the finite-state automaton (FSA) for construction of complex regular expressions for pattern matching. While FSA is a fundamental topic in Computer Science (CS) studies, this is seldom mentioned outside the CS circle. Although there is a high degree of correspondence between regular expressions and FSA, this may not be at all obvious to a reader without relevant background, despite I have avoided rigorous treatment of the topic and tried to explain it in a manner that would be more easily communicable to fellow readers. I would like to emphasize this topic is not essential in Perl programming, and I only intend to use it as a tool to formulate better patterns. Feel free to skip it if that is not comfortable to you and I require your feedback of whether these topics can help you as I originally intended.

It is important for me to reiterate that this document is not intended to be a substitute for the official Perl manual pages (aka *man pages*) and other official Perl literature. In fact, it is the set of manual pages that covers the Perl language in sufficiently fine detail, and it will be the most important set of document after you have accumulated certain level of knowledge and programming experience. The Perl man pages are written in the most concise and correct technical parlance, and as a result

they are not very suitable for new programmers to understand. The primary objective of this tutorial is to bridge the gap so as to supplement readers with sufficient knowledge to understand the man pages. Therefore, this tutorial presents a different perspective compared with some other Perl guidebooks available at your local bookstores from the mainstream computer book publishers. With a Computer Science background, I intend to go more in-depth into the principles which are central to the study of programming languages in general. Apart from describing the syntax and language features of Perl, I also tried to draw together the classical programming language design theories and explained how they are applied in Perl. With this knowledge, it is hoped that readers can better understand the jargons presented in manual pages and the principles behind. Perl is attributed by some as a very cryptic language and is difficult to learn. However, those who are knowledgeable about programming language design principles would agree Perl implements a very rich set of language features, and therefore is an ideal language for students to experiment with different programming language design principles taught in class in action. I do hope that after you have finished reading this tutorial you will be able to explore the Perl horizons on your own with confidence and experience the exciting possibilities associated with the language more easily. *"To help you learn how to learn"* has always been the chief methodology followed in this tutorial.

Time flies. Today when I am revising this preface, which was actually written before I made my initial preview release in late 2001 according to the timestamp, I am aghast to find that it has already been nearly two years since I started writing it. Indeed, a lot of things have changed in two years. Several Perl manpages written in tutorial-style have been included into the core distribution, which are written in a more gentle way targeted at beginners. There are also more Perl resources online today than there were two years ago. However, I believe through preparing this tutorial I have also learnt a lot in the process. Despite I started this project two years ago, a major part of the text was actually written in a window of 3 months. As a result, many parts of the tutorial were unfortunately completed in a hasty manner. However, through constant refinement and rewriting of certain parts of the tutorial, I believe the Second Edition will be more well-organized and coherent, while more advanced topics can be accommodated as more ample time is available.

At last, thank you very much for choosing this tutorial. Welcome to the exciting world of Perl programming!

Bernard Chan
in Hong Kong, China
15th September, 2003

Typographical Conventions

Although care has been taken towards establishing a consistent typographical convention throughout this tutorial, considering this is the first time I try to publish in LaTeX, slight deviations may be found in certain parts of this document. Here I put down the convention to which I tried to adhere:

Elements in programming languages are typeset in `monospace font`.

Important terms are typeset in **bold**.

Profound sayings or quotes are typeset in *italic*.

In source code listings, very long lines are broken into several lines. ¶ is placed wherever a line break occurs.

Release History

30th August, 2003	First Edition, Release Candidate 1
15th September, 2003	First Edition, Release Candidate 2
01st October, 2003	First Edition, Release Candidate 3
January, 2004	First Edition

Chapter 1

Introduction to Programming

1.1 What is Perl?

Extracted from the perl manpage,
"Perl is an interpreted high-level programming language developed by Larry Wall."

If you have not learnt any programming languages before, as this is not a prerequisite of this tutorial, this definition may appear exotic for you. The two keywords that you may not understand are "interpreted" and "high-level". Because this tutorial is mainly for those who do not have any programming experience, it is better for me to give you a general picture as to how a computer program is developed. This helps you understand this definition.

1.2 A Trivial Introduction to Computer Programming

You should know that, regardless of the programming language you are using, you have to write something that we usually refer to as **source code**, which include a set of instructions for the computer to perform some operations dictated by the programmer. There are two ways as to how the source code can be executed by the Central Processing Unit (CPU) inside your computer. The first way is to go through two processes, **compilation** and **linking**, to transform the source code into **machine code**, which is a file consisting of a series of numbers only. This file is in a format that can be recognized by the CPU readily, and does not require any external programs for execution. Syntax errors are detected when the program is being compiled. We describe this executable file as a **compiled** program. Most software programs (e.g. most EXEs for MS-DOS/Windows) installed in your computer fall within this type.

NOTES

There are some subtleties, though. For example, the compiler that comes with Visual Basic 6 Learning Edition translates source code into p-code (pseudo code) which has to be further converted to machine code at runtime. Such an EXE is described as interpreted instead. Therefore, not all EXEs are compiled.

On the other hand, although Java is customarily considered an interpreted language, Java source files are first compiled into bytecode by the programmer, so syntactical errors can be checked at compile time.

Another way is to leave the program uncompiled (or translate the source code to an intermediate level between machine code and source code, e.g. Java). However, the program cannot be executed on its own. Instead, an external program has to be used to execute the source code. This external program is known as an **interpreter**, because it acts as an intermediary to interpret the source code in a way the CPU can understand. Compilation is carried out by the interpreter before execution to check for syntax errors and convert the program into certain internal form for execution. Therefore, the main difference between compiled programs and interpreted languages is largely only the time of compilation phase. Compilation of compiled programs is performed early, while for interpreted programs it is usually performed just before the execution phase.

Every approach has its respective merits. Usually, a compiled program only has to be compiled once, and thus syntax checking is only performed once. What the operating system only needs to do is to read the compiled program and the instructions encoded can be arranged for execution by the CPU directly. However, interpreted programs usually have to perform syntax check every time the program is executed, and a further compilation step is needed. Therefore, startup time of compiled programs are usually shorter and execution of the program is usually faster. For two functionally equivalent programs, a compiled program generally gives higher performance than the interpreted program. Therefore, performance-critical applications are generally compiled. However, there are a number of factors, e.g. optimization, that influence the actual performance. Also, the end user of a compiled program does not need to have any interpreters installed in order to run the program. This convenience factor is important to some users. On the other hand, interpreters have to be installed in order to execute a program that is interpreted. One example is the **Java Virtual Machine** (JVM) that is an interpreter plugged into your browser to support Java applets. Java source files are translated into **Java bytecode**, which is then executed by the interpreter. There are some drawbacks for a compiled program. For example, every time you would like to test your software to see if it works properly, you have to compile and link the program. This makes it rather annoying for programmers to fix the errors in the program (**debug**), although the use of makefiles alleviates most of this hassle from you. Because compilation translates the source code to machine code which can be executed by the hardware circuitry in the CPU, this process creates a file in machine code that depends on the **instruction set** of the computer (machine-dependent). On the other hand, interpreted programs are usually **platform-independent**, that is, the program is not affected by the operating system on which the program is executed. Therefore, for example, if you have a Java applet on a Web site, it can most probably be executed correctly regardless of the operating system or browser a visitor is using. It is also easier to debug an interpreted program because repeated compilation is waived.

TERMINOLOGY

Instruction set refers to the set of instructions that the CPU executes. There are a number of types of microprocessors nowadays. For example, IBM-compatible PCs are now using the Intel-based instruction set. This is the instruction set that most computer users are using. Another prominent example is the Motorola 68000 series microprocessors in Macintosh computers. There are some other microprocessor types which exist in minority. The instruction sets of these microprocessors are different and, therefore, a Windows program cannot be executed unadapted on a Macintosh computer. In a more technical parlance, different microprocessors have different **instruction set architectures**.

Recall that I mentioned that a compiled program consists entirely of numbers. Because a CPU is actually an electronic circuit, and a digital circuit mainly deals with Booleans (i.e. 0 and 1), so it is

obvious that programs used by this circuit have to be sequences of 0s and 1s. This is what machine code actually is. However, programming entirely with numbers is an extreme deterrent to computer programming, because numeric programming is highly prone to mistakes and debugging is very difficult. Therefore, assembly language was invented to allow programmers to use mnemonic names to write programs. An assembler is used to translate the assembly language source into machine code. Assembly language is described as a low-level programming language, because the actions of an assembly language program are mainly hardware operations, for example, moving bits of data from one memory location to another. Programming using assembly language is actually analogous to that of machine code in disguise, so it is still not programmer friendly enough.

Some mathematicians and computer scientists began to develop languages which were more machine-independent and intuitive to programmers that today we refer to as **high-level programming languages**. The first several high-level languages, like FORTRAN, LISP, COBOL, were designed for specialized purposes. It was not until BASIC (Beginner's All-purpose Symbolic Instruction Code) was invented in 1966 that made computer programming unprecedentedly easy and popular. It was the first widely-used high-level language for general purpose. Many programmers nowadays use C++, another high-level language, to write software programs. The reason why we call these "high-level languages" is that they were built on top of low-level languages and hid the complexity of low-level languages from the programmers. All such complexities are handled by the interpreters or compilers automatically. This is an important design concept in Computer Science called **abstraction**.

That's enough background information and we can now apply the concepts learned above to Perl. Perl (Practical Extraction and Reporting Language) was designed by Larry Wall, who continues to develop newer versions of the Perl language for the Perl community today. Perl does not create standalone programs and Perl programs have to be executed by a Perl interpreter. Perl interpreters are now available for virtually any operating system, including but not limited to Microsoft Windows (Win32) and many flavours of Unix. As I quoted above, *"Perl is a language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information."* This precise description best summarizes the strength of Perl, mainly because Perl has a powerful set of **regular expressions** with which programmers can specify search criteria (patterns) precisely. You are going to see a whole chapter devoted to regular expression in Chapter 9. Perl is installed on many Web servers nowadays for dynamic Web CGI scripting. Perl programs written as CGI applications are executed on servers where the Perl source files are placed. Therefore, there is no need to transfer the Perl source to and from the server (as opposed to client-side scripts like JavaScript or Java applets). Guestbooks, discussion forums and many powerful applications for the Web can be developed using Perl.

There is one point which makes Perl very flexible — there is always more than one approach to accomplish a certain task, and programmers can pick whatever approach that best suits the purpose.

1.3 Scripts vs. Programs

There has always been some arguments over whether to use the term "script" or "program" for Perl source files. In general, a piece of code that is executed by hardware or a software interpreter, written in some kind of programming languages, is formally called a "program". This is a general term that applies to programs written in machine instructions, or any programs that are compiled or interpreted. However, it is also common today to hear that people use the term "script" to refer

to programs that are interpreted, especially those executed on the command line. In my opinion, it is not important to draw a distinction between the two terms as both are considered equally acceptable and understandable nowadays. In many situations people just use both interchangeably.

I am more inclined towards calling Perl programs and CGI programs running on a Perl backend as scripts, so I will adhere to this terminology in this tutorial.

1.4 An Overview of the Software Development Process

An intuitive software development process is outlined below. Note that this process is not tailored for Perl programming in particular. It is a general development process that can be applied to any programming projects with any programming languages. For additional notes specific to Perl, please refer to the next chapter.

Because this tutorial does not assume readers to have any programming experience, it is appropriate for me to give you an idea as to the procedure you will most probably follow when you write your programs. In general, the process of development of a software project could be broken down into a number of stages. Here is an outline of the stages involved:

- ⋆ **Requirements Analysis**
  First you need to identify the requirements of the project. Simply speaking, you will need to decide what your program should do (known as **functional requirements**), and note down other requirements that are important but not related to the functions of your program (known as **non-functional requirements**), for example, a requirement that the user interface should be user friendly. You have to make a list of the requirements, and from it you will need to decide whether you have the capability to complete them. You may also want to prioritize them such that the most important functionalities are developed first, and other parts can be added subsequently.

- ⋆ **Systems Design**
  From the requirements determined you can then define the scope of the project. Instead of putting the whole program in one piece, we will now organize the program into several components (or **subsystems** — a part of the entire system). As we will discuss later in this tutorial, modularization facilitates code reuse and make correction of bugs (debug) easier. Two major models exist today — decomposition based on functions and decomposition based on objects. After you have fixed the model, you decide on which functions or object methods are to be associated with which source file or object, and determine how these components interact with each other to perform the functionalities. Note that you don't need to decide on how these source files or objects are implemented in real source code at this stage — it is just an overall view of the interaction between the components. We emphasize functional decomposition in the first part of the tutorial, while object-oriented programming will be covered in a later part of this tutorial.

- ⋆ **Program Design**
  After we have determined how the components interact with each other, we can now decide how each function or object method is implemented. For each function, based on the actions to perform you have to develop an **algorithm**, which is a well-defined programming-language-independent procedure to carry out the actions specified. You may want to use a flowchart or some **pseudocode** to illustrate the flow of the program. Pseudocode is expressed

in a way resembling real programming source code, except language-dependent constructs are omitted. As pseudocode is language independent, you can transform an idea from pseudocode to source code in any programming languages very easily. There isn't a single standardized pseudocode syntax. In many cases, pseudocode can even be written in English-like statements because pseudocode is written to demonstrate how a program is supposed to work, and provided it communicates the idea clearly it suffices. It is up to you as the author to express pseudocode in whatever way the algorithm is best illustrated.

- ⋆ **Coding**
  This is largely the continuation of the Program Design stage to transform your algorithm into programming language constructs. If you have worked out the algorithm properly this should be a piece of cake.

- ⋆ **Unit Testing**
  Unit testing corresponds to Program Design. As each function or object method has a predefined behaviour, they can be tested individually to see if such behaviour agree to that defined. Most of the time when we are talking about debugging, we are describing this stage.

- ⋆ **Systems Testing**
  Systems Testing corresponds to System Design. This is to test if the components interact with each other in exactly the same way as designed in the Systems Design stage.

- ⋆ **Requirements Validation**
  This corresponds to requirements analysis. The software developed is compared against the requirements to ensure each functionality has been incorporated into the system and works as expected.

- ⋆ **Maintenance**
  By now the software has been developed but you cannot simply abandon it. Most probably we still need to develop later versions, or apply patches to the current one as new bugs are found in the program. Software for commercial distribution especially needs investment of a lot of time and effort at this stage capturing user feedback, but software not distributed commercially should also pay attention to this stage as this affects how well your software can be further developed.

Of course, for the examples in this tutorial that are so short and simple we don't need such an elaborate development procedure. However, you will find when you develop a larger-scale project that having a well-defined procedure is essential to keep your development process in order.

This is just one of the many process models in existence today. Discussion of such process models can be found in many fundamental text for **Software Engineering**, and are beyond the scope of this tutorial. Actually, what I have presented was a variant of the Waterfall process model, and is considered one that, if employed, is likely to delay project schedules and result in increased costs of software development. The reason I present it here is that the Waterfall model is the easiest model to understand. Because presentation of process models is out of the scope of the tutorial, some Web links will be presented at the end of this chapter from which you will find selected texts describing process models, including the **Rational Unified Process** which I recommend as an improved process model for larger-scale development projects. Adoption of an appropriate process model helps guide the development process with optimized usage of resources, increased productivity and software that are more fault-tolerant.

Summary

- Source code include a set of instructions to be executed by the computer through levels of translation.
- Low-level programming languages involves architecture-dependent programming with machine code or assembly language.
- High-level programming languages are built on top of low-level languages and source programs can usually be platform-independent.
- Source programs are either explicitly precompiled into object code, or are implicitly compiled before execution by the interpreter.
- Perl is an interpreted high-level programming language developed by Larry Wall.
- Following a well-defined software development process model helps keep the development process systematic and within budgets.

Web Links

1.1 Evolution of Programming Languages
http://lycoskids.infoplease.com/ce6/sci/A0860536.html

1.2 Rational Unified Process Whitepapers
http://www.rational.com/products/rup/whitepapers.jsp

Chapter 2

Getting Started

2.1 What can Perl do?

I understand it is a full wastage of time for you to have read through half of a book to find that it is not the one you are looking for. Therefore, I am going to let you know what you will learn by following this tutorial as early as possible.

If you are looking for a programming language to write an HTML editor that runs on the Windows platform, or if you would like to write a Web browser or office suite, then Perl does not seem to be an appropriate language for you. C/C++, Java or (if you are using Windows) Visual Basic are likely to be more appropriate choices for you.

Although it appears that Perl is not the optimum language for developing applications with a graphical user interface (but you can, with Perl/Tk or native modules like `WIN::GUI`), it is especially strong in doing text manipulation and extraction of useful information. Therefore, with database interfacing it is possible to build robust applications that require a lot of text processing as well as database management.

Perl is the most popular scripting language used to write scripts that utilize the **Common Gateway Interface** (CGI), and this is how most of us got to know this language in the first place. A cursory look at the CGI Resource Index Web site provided me with a listing of about 3000 Perl CGI scripts, compared with only 220 written in C/C++, as of this writing. There are quite many free Web hosts that allow you to deploy custom Perl CGI scripts, but in general C/C++ CGI scripts are virtually only allowed unless you pay. In particular, there are several famous programs written in Perl worth mentioning here:

- ⋆ YaBB is an open source bulletin board system. While providing users with many advanced features that could only be found on commercial products, it remains as a free product. Many webmasters use YaBB to set up their BBS. Another popular BBS written in Perl is ikonboard, featuring a MySQL/PostgreSQL database back-end.

- ⋆ Thanks to the powerful pattern matching functions in Perl, search engines can also be written in Perl with unparalleled ease. Perlfect Search is a very good Web site indexing and searching system written in Perl.

You will learn more about Perl CGI programming in Chapter 11 of this tutorial.

2.2 Comparison with Other Programming Languages

There are many programming languages in use today, each of which placing its emphasis on certain application domains and features. In the following section I will try to compare Perl with several popular programming languages for the readers to decide whether Perl is appropriate for you.

2.2.1 C/C++

Perl is written in the C programming language. C is extensively used to develop many system software. C++ is an extension of C, adding various new features such as namespaces, templates, object-oriented programming and exception handling etc. Because C and C++ programs are compiled to native code, startup times of C/C++ programs are usually very short and they can be executed very efficiently. Perl allows you to delegate part of your program in C through the Perl-C XS interface. This Perl-C binding is extensively used by cryptographic modules to implement the core cryptographic algorithms, because such modules are computation-intensive.

While C/C++ is good for performance-critical applications, C/C++ suffers a number of drawbacks. First, C/C++ programs are platform dependent. A C/C++ program written on Unix is different from one on Windows because the libraries available on different platforms are different. Second, because C/C++ is a very structured language, its syntax is not as flexible as scripting languages such as Perl, Tcl/Tk or (on Unix platforms) bash. If you are to write two functionally equivalent programs in C/C++ and Perl, very likely the C/C++ version requires more lines of code compared with Perl. And also, improperly written C/C++ programs are vulnerable to memory leak problems where heap memory allocated are not returned when the program exits. On a Web server running 24×7 with a lot of visitors, a CGI script with memory leak is sufficient to paralyze the machine.

2.2.2 PHP

Perl has been the traditional language of choice for writing server-side CGI scripts. However, in recent years there has been an extensive migration from Perl to PHP. Many programmers, especially those who are new to programming, have chosen PHP instead of Perl. What are the advantages of PHP over Perl?

PHP is from its infancy Web-scripting oriented. Similar to ASP or JSP, it allows embedding of inline PHP code inside HTML documents that makes it very convenient to embed small snippets of PHP code, e.g. to update a counter when a visitor views a page. Perl needs Server Side Includes (SSI) or an additional package "eperl" to implement a similar functionality. Also, it inherits its language syntax from a number of languages so that it has the best features of many different languages. It mainly inherits from C/C++, and portions from Perl and Java. It uses I/O functions similar to that in C, that are also inherited into Perl, so it is relatively easy for Perl programmers to migrate to PHP.

While PHP supports the object-oriented paradigm, most of its functionalities are provided through functions. When PHP is compiled the administrator decides the sets of functionalities to enable. This in turn determines the sets of functions enabled in the PHP installation. I'm personally sceptical of this approach, because in practice only a small subset of these functions is frequently used. On the other hand, Perl only has a small set of intrinsic functions covering the most frequently used functionalities. Other functionalities are delegated to modules which are only installed and invoked as needed. As I will introduce shortly and in Appendix B, the Comprehensive Perl Archive Network (CPAN) contains a comprehensive and well-organized listing of ready-made Perl modules that you

can install and use very easily.

2.2.3 Java/JSP

Sun Microsystems developed the Java language and intended to target it as a general purpose programming language. It is from the ground up object-oriented and platform independent. Functionalities are accessed through the Java API, consisting of hierarchies of classes similar to that of Perl. Java Server Pages (JSP) is a Web scripting environment similar to ASP except with a Java syntax. Similar to C/C++, the Java syntax is very structured and thus are not as flexible as scripting languages like Perl. Also, Java itself is not just an interpreter, it is a virtual machine over which programmers are totally abstracted from the underlying operating system platforms, which allows the Java API to be implemented on top of this platform-independent layer. For those who have programmed in Java before, you will probably find that the Java Virtual Machine takes rather long time to load, especially on lower-end systems with limited computational power. This defers the possibility of widespread deployment of Java programs.

While Perl is not strictly a general-purpose programming language like Java, I found it difficult to compare Perl and Java because of their different natures. However, if confined to the purpose of Web server scripting, I generally prefer Perl to JSP for its flexibility and lightweight performance. Despite this, I feel that Java is a language that is feature-rich and if time allows, you are strongly encouraged to find out more about this stupendous language, which is expecting increasing attention in mobile and embedded devices because of its platform independence.

2.2.4 ASP

Active Server Pages (ASP) is only available on Windows NT-series operating systems where Internet Information Services (IIS) is installed (although alternative implementations of ASP on other system architectures exist, e.g. Sun Chili!Soft ASP, which is a commercial product that runs on Unix, but generally considered not very stable).

Running on a Windows Web server, ASP can impose a tighter integration with Microsoft technologies, so that the use of, say, ActiveX data objects (ADO) for database access can be made a lot easier. However, IIS is especially vulnerable to remote attacks when operated as a Web server. Numerous service packs have been released to patch the security holes in IIS and Windows NT. However, new holes are still being discovered from time to time that makes the deployment of Windows NT/IIS as the Web server of choice not very favourable. On the other hand, Apache, the renowned Web server for Unix and now for other operating systems as well, has far less security concerns and are less susceptible to remote attacks. Apache also has the largest installation base among all Web server software, taking up more than 60% of the market share.

2.3 What do I need to learn Perl?

You don't need to pay a penny to learn and use Perl. Basically, a text editor that handles text-only files and a working installation of the Perl interpreter are all that you will need.

Under Microsoft Windows, Notepad meets the minimum requirement. However, a whole page of code in black is not visually attractive in terms of readability. Some text editors have the feature

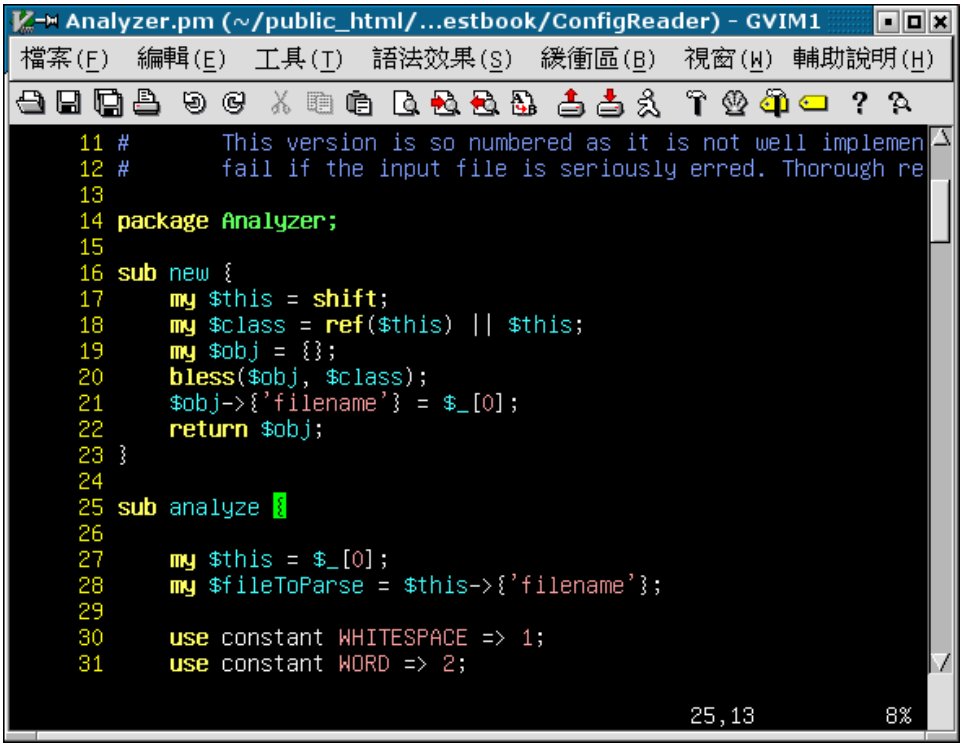

Figure 2.1: Editing a Perl source file with GVIM, running on GNU/Linux

of **syntax highlighting**, with different parts of a statement displayed in different colours. Good colouring makes the source files more pleasurable to look at (such colouring is used for display only and will not be saved to file). However, avoid using word processors like Microsoft Word or Wordpad which add proprietary control codes on file save by default. The Perl interpreter does not recognize these special formats. If you have to use these word processors, ensure that your files are saved as plain text ASCII format so that the Perl interpreter can access them. AnyEdit and UltraEdit are nice text editors on the Windows platform. On Unix variants, emacs and vim are stupendous text editors featuring syntax highlighting profiles for most programming languages with a lot of powerful features. Fig. 2.1 shows a screenshot of a Perl source file edited with GVIM, a port of vim that runs on Windows, X-Windows with the GTK library on Unix/Linux and many other platforms. This is my favourite text editor and is used to construct my entire Web site.

If you are using one of the mainstream operating systems, the perl interpreter can be downloaded from the download section of perl.com. perl.com is the official Web site for the Perl language and you can find the download links to all available interpreter versions there. Choose the version which matches your operating system. When you go to the download page you will see two versions, namely the stable production release and the experimental developer's release. The stable release is the version I recommend to new users, because the developer's version is for more advanced users to beta test the new version. It may still contain bugs and may give incorrect results. The files you have to download are under the heading "binary distribution". Do not download the source code distribution unless you know exactly how to compile and install them. In case you are using an operating system that is not listed, a good place to find a binary distribution for your operating system is the CPAN, located at here, which contains a fairly comprehensive list of platforms on which Perl can run.

For Windows users, most probably you should download the Activestate distribution of Perl. It is very easy to install, with some extra tools bundled in the package for easy installation of new

modules. For GNU/Linux users, most probably Perl is already installed or available as RPM (Redhat Package Manager) or DEB (Debian packages) formats. As many Linux distributions already have builtin support for RPM packages, you may look at your installation discs and you are likely to find some RPM binaries for Perl if it is not yet installed. For other Unix systems, you may find tarballs containing the Perl binaries. If no binaries are available for your system, you can still build from sources by downloading the source distribution from the CPAN. To check if perl is installed on your system, simply open a terminal and type `perl -v`. If Perl is installed you will have the version information of Perl installed displayed on screen. If error messages appear, you will need to install it.

Installation of Perl will not be covered in this tutorial, and you should look for the installation help documents for details.

NOTES

Because Perl is an open source software, which releases the source code to the public for free, you will see the source code distribution listed. Yet for usual programming purposes there is no need to download the source files unless binary distributions are not available for your system.

An exception is if you are using one of the operating systems in the Unix family (including Linux). There are already compilation tools in these operating systems and you can manually compile Perl from sources and install it afterwards. However, note that compilation can be a very time-consuming process, depending on the performance of your system. If you are using Linux, binary distributions in the form of RPM or DEB packages can be installed very easily. Only if you cannot find a binary distribution for your platform that you are encouraged to install from source package.

2.4 Make Good Use of Online Resources

You may need to seek reference while you are learning the language. As a new user you are not recommended to start learning Perl by reading the man-pages or the reference manuals. They are written in strict technical parlance that beginners, especially those who do not have prior programming experience or basic knowledge in Computer Science, would find reading them real headaches. You are recommended to follow this tutorial (or other tutorials or books) to acquire some basic knowledge first, and these reference documents will become very useful for ambitious learners to know more about the language, or when you have doubt on a particular subject you may be able to find the answers inside. In this course I will try to cover some important terms used in the reference materials to facilitate your understanding of the text. For the time being, you may want to have several books on Perl for cross-referencing purposes. I have tried to write this tutorial in a way that beginners should find it easy to follow, yet you may need to consult these books if you have any points that you don't understand fully.

Although you are not advised to read the official reference documents too early, in some later parts I may refer you to read a certain manpage. A manpage, on Unix/Linux systems, is a help document on a particular aspect. To read a particular manpage, (bring up a terminal if you are in X-Windows) type `man` followed by the name of the manpage, for example, `man perlvar`, the perlvar manpage will be displayed. For other platforms, manpages may usually come in the format of HTML files. Consult the documentation of your distribution for details. There is an online

version of the Perl official documentation at perldoc.com. It contains the Perl man pages as well as documentation of the modules shipped with Perl. In fact, there are now several manpages that are targeted at novice programmers. For instance, the perlintro manpage is a brief introduction to the fundamental aspects of the Perl language that you should master fully in order to claim yourself a Perl programmer.

You are also reminded of the vast varieties of Perl resources online. There are many Perl newsgroups on the USENET and mailing lists devoted to Perl. Your questions may be readily answered by expert Perl programmers there. Of course, try to look for a solution from all the resources you can find including the FAQs before you post! Otherwise, your question may simply be ignored. Perl Monks is also a very useful resource to Perl users.

dmoz.org contains a nice selection of Perl-related sites. You can find a similar list of entries on Yahoo!.

Google is the best search engine for programmers. You can usually get very accurate search results that deliver what you need. For example, by specifying the terms *"Perl CGI.pm example"*, you will get screenfuls of links to examples demonstrating the various uses of the CGI.pm module. As you will see later, this module is the central powerhouse allowing you to handle most operations involving CGI programming with unparalleled ease. For materials not covered in this tutorial or other books, a search phrase can be constructed in a similar manner that allows you to find documentation and solutions to your questions at your fingertips.

Of course, don't forget to experiment yourself! CPAN, the Comprehensive Perl Archive Network is a nice place where you can download a lot of useful modules contributed by other Perl programmers. By using these modules you can enforce code reuse, rather than always inventing code from scratch again. There are so many modules on the CPAN available that you would be surprised at how active the Perl community has been. Some CPAN modules are well-documented, some are not. You may need to try to fit the bits and pieces together and see if it works. This requires much time and effort, but you can learn quite a lot from this process.

2.5 The Traditional "Hello World" Program

Traditionally, the first example most book authors use to introduce a programming language is what is customarily called a "Hello World" program. The action of this program is extremely simple — simply displays the text "Hello World" to the screen and does nothing else. For all examples in this tutorial of which source code are given in the text, you are encouraged to type them in yourself instead of executing the examples downloaded from my Web site, since it is more likely that by doing so you would understand the materials more quickly. Let's write a "Hello World" program to see the procedures we take to develop a Perl program.

If you are on Windows, it is a good practice to check if the path to the Perl interpreter has been added to the path list in C:\Autoexec.bat. In this way, you can change to the path containing your Perl source files and can run the interpreter without specifying its path. The setup program of your distribution would probably have done it for you. If it hasn't, append the path to the end of the list and end it with a semicolon. A typical path list looks like this, the last one in this example is the path to the perl interpreter (note that your path may be different):

```
SET PATH = C:\WINDOWS; C:\WINDOWS\COMMAND; C:\WINDOWS\SYSTEM; C:\PERL\BIN;
```

NOTES

The use of Autoexec.bat is now obsolete starting from Windows 2000. Setting of environment variables should be carried out by right-clicking on the "My Computer" icon, and then choose the "Properties" option. Now select the "Advanced" tab and then click on the "Environment Variables" button at the bottom. To make the perl interpreter available to all users on the system, the path should be appended to the PATH variable in the "System variables" section. If you modify the PATH variable in the "User variables" section, only the user concerned (presumably you) will be affected.

For Unix/Linux, check your PATH variable and see if the directory containing the perl executable is present (usually /usr/bin). You can look at the list of paths by typing `echo $PATH` on the command line (be careful of exact capitalization!). Look for "/usr/bin" in the colon-separated values. On some systems, the path to perl would be "/usr/local/bin" or something else, so please check carefully. You may need to modify the startup login scripts like .login, .bashrc, .profile etc. so that you don't need to set PATH or specify the full path to perl every time if perl is installed at some weird locations. A convenient workaround is to create a symbolic link in a directory included in PATH, e.g. /usr/bin that points to the perl executable.

EXAMPLE 2.1

```
1  #!/usr/bin/perl -w
2  # Example 2.1 - Hello World
3
4  # print the text to the screen
5  print "Hello, World!\n";
```

Here we outline the steps to create this simple program on Windows and Linux.

Microsoft Windows

1. Open Notepad (or any other text editor you choose) and type in the source code shown above. Note that the line numbers on the left are for identification of lines only and do NOT type them into the text editor. Please make sure word wrap is disabled.

2. Save the file as hello.pl. A few text editors, like Notepad, usually append the ".txt" extension to the filename when saving. You may put a pair of double quotes around the filename to circumvent this behaviour. Also, if you are using Windows 2000 or above and would like to use Notepad, please ensure that the file encoding is set to ANSI instead of Unicode.

3. Bring up an MS-DOS prompt window and change to the directory containing your newly created file. Say if you have saved to "C:\perl_examples", then type `cd C:\perl_examples` and press Enter. Put a pair of double quotes around the path if any directories in the path contains spaces (In fact I don't recommend placing Perl source files in directories with names containing spaces. It only complicates matters).

4. Execute the program by typing `perl -w hello.pl` and press enter.

Unix or GNU/Linux

1. Open any text editor (vim, emacs, pico, kedit ....) and type in the source code shown above. Note that the line numbers on the left are for identification only and do NOT type them into the text editor. Please make sure word wrap is disabled.

2. Save the file as hello.pl. Note that the path on line 1 has to match the path to perl on your system. Also, no spaces should precede the '#' character and no empty lines are allowed before this special line (traditionally known as the '**shebang**' line).

3. If you are in X-Windows environment, bring up a terminal window. Change to the directory containing the newly created file using the `cd` command.

4. In order to run it without specifying the perl interpreter, set the file access privilege to user executable by using the `chmod` command. The command should be `chmod u+x hello.pl` and press Enter.

5. Execute the program by typing `./hello.pl` and then press Enter.

NOTES

Even if you are using Unix/Linux, it is not absolutely needed to chmod your perl source files. In fact, you only need to make those source files executable if you want them to be directly invoked without specifying the name of the interpreter (i.e. `perl`). In this case, the shell will look at the first line of the file to determine which interpreter is used, so the `#!/usr/bin/perl` line must exist. Otherwise, the file cannot be executed. If you only intend to execute the file in Unix or Linux using `perl -w filename.pl`, then filename.pl need not be given an executable permission. As you will learn later, you may have some Perl source files that are not invoked directly. Instead, they are being sourced from another source file and don't need to be executable themselves. For these files, you don't need to chmod them and the default permission is adequate.

If there is not any errors, you should see the words "Hello, World!" under the command prompt. For such a simple program it is not easy to make mistakes. If error messages appear, check carefully if you have left out anything, because a trivial mistake is sufficient to end up with some error messages if you are not careful. Also check if you are using the latest stable version of Perl 5. All examples in this tutorial have been tested with Perl 5.8.0 Win32 (ActiveState distribution) and Perl 5.8.0 on GNU/Linux, but it should work for other distributions or versions as well unless otherwise noted.

The `-w` is an example of a **switch**. You specify a switch to enable a particular interpreter feature. `-w` is specified so that warning messages, if any, are displayed on screen. Under no circumstances should this switch be omitted because it is important, especially as a beginner, to ensure that the code written is correct. It also helps catch some mistakes that are otherwise difficult to capture. This is explained in more detail in Chapter 10.

A Perl script consists of **statements**, and each statement is terminated with a semicolon (;). A statement is rather like a sentence in human languages which carries a certain meaning. For computer languages a statement is an instruction to be performed.

The core of the program is on line 5. It is this statement that prints the text delimited by quotation marks to the screen (in a more accurate parlance, the text is sent to the **standard output**, which is the screen by default). `print()` is an example of a builtin **function**. A function usually accepts a number of **parameters**, or **arguments**. Parameters serve to provide a function with additional pieces of data that are necessary for its operation. In the example, `print()` takes the text as parameter in order to display it on screen. A set of basic functions is provided to you by Perl for performing different actions. Some of these functions will be introduced as you progress through this tutorial.

Notice the strange `\n` at the end? It is one of the **escape characters** which will be described later in more detail. `\n` is used to insert a line break. Therefore, you see a blank line before returning to the command prompt.

Lines preceded by a # (sharp) sign are **comments** and are ignored by the perl interpreter. A comment does not need to be on its own line, it can be put at the end of a line as well. In that case, the remaining of the line (starting from # and until the end of the line) is regarded as a comment. Comments are helpful for you or other programmers who read your code to understand what it does. You can put anything you like as comments. Line 1 is also a comment as it is of interest to the shell only instead of the perl interpreter itself. The switch `-w` here is the same as that specified under the command line. This is read together with the path to perl to enable the display of warnings.

GOOD PROGRAMMING PRACTICES

Comments

Comments are meant for human readers to understand the source code without the need of running the program once in your brain. This increases both the readability and maintainability of your source code. Many programmers are lazy to insert comments throughout the code. But it is very likely when you look at a piece of code wrote earlier you may not understand it anymore as it is very complicated without any comments in it.

Therefore, you should include comments in appropriate places. Usually for a single block of code performing one particular function we will place a comment briefly describing what this code block does. You may also want to place a comment on a particular line if the meaning of the line is not immediately obvious. Of course, don't deluge your source code with comments. For example, in the source code of the 'Hello World' program I placed a comment for the `print` statement. It is superfluous, in fact, as the meaning of this statement is pretty obvious. But I included it here because this is your first Perl function learnt. As you proceed, more constructive comments and less superfluous comments will be found in the examples.

2.6 How A Perl Program Is Executed

Perl programs are distributed in source files. From the instance you invoke the perl interpreter to execute a script, a number of steps were involved before the program is executed.

Preprocessing An optional preprocessing stage transforms the source file to the final form. This tutorial does not cover source preprocessing. For details please consult the perlfilter manpage.

Tokenizing The source file is broken down into tokens. This process is called tokenization (or **lexical analysis**). Whitespaces are removed. Token is the basic unit that makes up a statement. By tokenizing the input parsing is becoming easier because all further processing are carried out on tokens, independent of whitespace.

Parsing & Optimization (Compilation) Parsing involves checks to ensure the program being executed conforms to the language specification and builds a **parse tree** internally which describes the program in terms of microoperations internal to Perl (opcode). Some optimizations to the parse tree are performed afterwards. Finally, the parse tree built is used to execute the program.

While in-depth understanding of any of these processes is not essential to practical Perl programming, the compilation phase will be mentioned in some later chapters that I believe it is a good idea to briefly introduce the phases involved beforehand.

2.7 Literals

All computer programs have to handle data. In every program there are certain kinds of data that do not change with time. For example, consider a very much simplified CGI script that checks if the password input by the user matches the system. How would you implement it? It seems the simplest method would be to have the correct password specified in the script, and after the user has entered the password and hit the "Submit" button, compare it against the password input by the user. The standard password specified in this script does not change during the course of execution of the script. This is an example of a **literal**. Other terms that are also used are **invariants** and **constants**. In the previous Hello World example, the text "Hello, World!\n" on line 5 is also a literal (This piece of data cannot be changed during the time you are running the program).

Literals can have a number of forms, just because we can have data of different forms. In Perl we can roughly differentiate numbers and strings.

2.7.1 Numbers

There are several classes of numbers: **integers** and decimals (known as **floating-point numbers** in computer literature).

In Perl, integers can be expressed in decimal (base 10), hexadecimal (base 16) or octal (base 8) notation. Octal numbers are preceded by a `0` (zero), e.g. `023` is equivalent to 19_{10} (the subscript 10 denotes representation in base-10, i.e. decimal form); hexadecimal numbers are preceded by `0x` (zero x), e.g. `0xfe` is equivalent to 254_{10}. For hexadecimal digits A - F, it does not matter whether you specify them in lowercase or uppercase. That is, `0xfe` is the same as `0xFE`.

Integers cannot be delimited by commas or spaces like 10,203,469 or 20 300. However, Perl provides a nice workaround as a substitute. An example is 4_976_297_305. This is just a facility to make large

numbers easier to read by programmers, and writing 4976297305 is entirely correct in Perl.

Decimals are those carrying decimal points. If the integral portion is 0, the integral portion is optional, i.e. -0.6 or -.6 work equally fine. Exponents (base 10) can also be specified by appending the letter "e" and the exponent to the real number portion. e.g. 2e3 is equivalent to $2 \times 10^3 = 2000$.

2.7.2 Strings

A **string** is a sequence of characters enclosed (delimited) by either double quotes (") or single quotes ('). They differ in variable substitution and in the way escape characters are handled. The text "Hello, World!\n" in the hello world example is a string literal, delimited by double quotes.

We will defer variable substitution until we come to variables. Escape characters exist in many programming languages. An escape character consists of a backslash \ symbol followed by a letter. Every escape character has a function associated with it. Escape characters are usually put inside double-quoted strings. Table 2.1 summarizes the most important escape characters used in Perl.

These escape characters are predefined and can be used in double-quoted strings. There is another case where backslashes have to be used in a string. This is known as character escaping. What does that mean? Consider you would like to use double quotes in a double-quoted string. For example you would like to print this English sentence instead of the "Hello World" phrase in Example 2.1:

Howdy says, "Give me $500".

That is, you try to `print` this sentence by:

```
print "Howdy says, "Give me $500".";
```

If you execute this statement, you will get into trouble, because " is used to mark the beginning and the end of the string literal itself. Perl locates the end of the string by searching forward until the second double quote is found. If the literal contains double quotes itself, Perl will not know where the string literal terminates. In the example above, Perl will think the string ends after "Howdy says, ". Also, after you have learned variable substitution in the next chapter you will realize that the symbol $ is used for variable substitution. You have to tell Perl explicitly you would like to use the symbol as is instead of performing variable substitution. To get around this problem, just place the \ character before the two symbols concerned, and this is what we mean to "escape" a character. So, the correct way to `print` this sentence using double quotes is:

```
print "Howdy says, \"Give me \$500\".";
```

However, wise Perl programmers will not do this, as the backslashes make the whole expression ugly. If we choose to use single quotes instead, we don't even have to escape anything:

```
print 'Howdy says, "Give me $500".';
```

Single-quoted strings do not support variable substitution, so the $ needs not be escaped. Also, because the symbol " does not carry any significance in the string, it does not need to be escaped as well. There are only two characters that need to be escaped in single-quoted strings, namely '

Escape Character	Function
\n	Newline Starts a newline
\r	Carriage Return Returns to the starting point of the line
\t	Tab Analogous to striking the **Tab** key on your keyboard; However, using tab to make formatter output does not always generate the format expected.
\b	Backspace Analogous to the **Backspace** key; erases the last character
\a	Bell Creates a beep sound from the system buzzer (or sound card)
\xnn	ASCII character using hexadecimal notation Outputs the character which corresponds to the specified ASCII index (each n is a hexadecimal digit)
\0nn	ASCII character using octal notation Outputs the character which corresponds to the specified ASCII index (each n is an octal digit)
\cX	Control Character For example, \cC is equivalent to pressing Ctrl-C on your keyboard
\u	Next letter uppercase The letter immediately following \u is converted to uppercase. For example, \uemail is equivalent to Email
\l	Next letter lowercase The letter immediately following \l is converted to lowercase. For example, \lEmail is equivalent to email
\U	All subsequent letters uppercase All the letters immediately following \U are converted to uppercase until \E is reached
\L	All subsequent letters lowercase All the letters immediately following \L are converted to lowercase until \E is reached
\Q	Disables pattern matching until \E This would be covered in the "Regular Expressions" chapter.
\E	Ends \U, \L, \Q Terminates the effect of \U, \L or \Q.

Table 2.1: The most commonly used escape characters in Perl

and \. For double-quoted strings, a number of characters have to be escaped, and it would become clear as you work through the chapters in this tutorial.

Empty strings are denoted by "" or '', that is, two quotes with nothing in between.

2.8 Introduction to Data Structures

Every programming language has certain kinds of data structures builtin. A data structure can be thought of as a virtual container residing in the memory in which data is stored. Each data structure is associated with a data type specifying the type of data permitted in the data structure. Data type is important in programming languages because data of different types are likely to be treated differently. For example, numbers are sorted by numerical value; while strings are sorted in alphabetical order. Many programming languages, like C++, Java and Visual Basic, have a large number of data types, e.g. integer, double, string, boolean ... just to name a few. They require declaration of a data type when a data structure is created, and this data type cannot be changed afterwards. There are both advantages and disadvantages to this approach. As different data types occupy different amount of storage in the memory, the underlying machine actually requires some type information to convert the high-level programming constructs into assembly instructions in the compilation stage. Also, by having the data type fixed there are less ambiguities as to how the data is to be handled. The most obvious disadvantage, of course, is that explicit data conversion is necessary in such programming languages.

As of Perl 5, Perl officially differentiates only two types of data: scalar data and list data. Moreover, Perl does not enforce strict type-checking, instead, it is loosely-typed. This may change in Perl 6, but you will not see it in the near future.

Scalar data represents a piece of data. All literals are scalar data. Variables are also scalar data. As the underlying machine requires explicit declaration of data types, Perl needs to convert the data between different data types as needed in the underlying implementation, while a Perl programmer can be oblivious to such data conversion. In the next chapter you would see example code in practice.

Another type is **list data**. List data is an aggregation of scalar data. Arrays and hashes belong to this type. While you may not have a clear picture of how list data look like at this point, you would have a clear idea after reading the next chapter.

Three basic data structures are provided by Perl, namely scalar variables, arrays and associative arrays (hashes). I am going to give an introduction to the three types of data structures at this point, and in the next chapter you would see the functions and operations associated with these data structures.

A **scalar variable**, or simply a variable, is a named entity representing a piece of scalar data of which the content can be modified throughout its lifetime. What does this mean? A variable is conceptually like a virtual basket, and only one object is allowed in it at any one time. If at some time you would like to place something else in the basket, you have to replace the existing object with a new one, and the existing object is discarded. In Perl a variable can store a piece of string or number (or a reference, which we haven't come to yet). Unlike other programming languages, Perl gives you the flexibility that at one time you may store a number and at other times you may store a string in the same variable, however, it is a good practice to always store data of a particular type

at any time in the same variable to avoid confusion.

Because the value of a variable can be modified at any point, and there can be many variables that are concurrently in use at a time, we have to specify which one to address. Therefore, variables are named, and on the other hand literals are unnamed. This name is known as an **identifier**. By default, all variables are **global**, that is, after the variable is first used in the script, it can be referred to at any time, anywhere until the script terminates. However, it is a good practice not to use global variables excessively. Instead, most variables are actually used for temporary storage only and can be restricted to be valid for a limited time. This concerns the lifetime of a variable, and in turn the idea of **scope**. This would be discussed in Chapter 5.

Sometimes we are dealing with a set of related data. Instead of storing them separately in variables, we may store them as list data, which is a collection of scalar data sharing a single identifier (name). Arrays and hashes are two types of list data.

An **array** is a named entity representing a list of scalar data, with each item assigned an **index**. In an array, the integral index (or **subscript**) uniquely identifies each item in the array. The first item has index 0, the one afterwards has index 1, and so on. Each item in an array is a piece of scalar data, and, therefore, (in Perl only) numbers as well as strings may coexist in the array. An array can be empty, that is, containing no elements (called a **null array**). A representation of an example array containing some data is shown below, the column on the left is the index and the data is on the right:

Index	Data
0	"Apple"
1	36
2	"Hello, World"
3	"School"

Table 2.2: Contents of a sample array in Perl

A **hash** is a special data structure. It is similar to an array except that the index is not an integer, so the term "index" is not customarily used for hashes. Instead, a string is used for indexing, and is known as a **key**. The key is conceptually like a tag which is attached to the corresponding value. The key and the value forms a pair (key-value pair). Like an array, the keys in a hash have to be distinct to distinguish a key-value pair from another. Recall that ordering in arrays is determined by the indices of the items (because we can say the first item is the one which has subscript 0, the second item which has subscript 1, and so on). However, in a hash no such ordering is present. You will see in the next chapter that we may "sort" the hashes by keys or by values for presentation purposes. However, this does not physically reorder the keys or the values in the hash. It merely rearranges the key-value pairs for screen output or to be passed to another process for further processing.

Hashes (or hash tables in Computer Science parlance) are especially useful in dictionary programs. Assume that the program works as follows. It requires a user to enter an English word into the text entry box that is to be searched in the dictionary database. Inside the dictionary is actually a long list of key-value pairs, where the key is the word entry and the value is an ID that the database uses internally to retrieve the corresponding record (containing the explanations, pronunciation etc.). The term entered by the user is queried in the dictionary. If the entry matches any key, the corresponding ID is obtained and is used to retrieve the record for the word specified; Otherwise, the term is not found and the program returns an error. Hash table is an efficient data structure for data storage. A well-implemented hash table requires only several comparisons to retrieve the

value if the key is in the hash. More surprisingly, even if a given key does not exist in a hash, it is NOT necessary to search through all the keys in the hash before returning the key-not-found error. The reason for this concerns the principle behind hash tables. You may find more information in Appendix A or in any textbooks on data structures and algorithms.

Several entries of a possible hash table for the above dictionary program is shown below:

Key	Value
"Boy"	342
"Apple"	165
"Kite"	1053
...	...

Table 2.3: Contents of a sample hash in Perl

In the next chapter, you will learn how to manipulate the data structures discussed. You will know how to construct an array, and remove items from it, etc.

Summary

- Perl is especially strong in text manipulation and extraction of information.
- To create a Perl program, you need only a text editor and the perl interpreter.
- The `-w` switch of the interpreter enables the output of warnings.
- The `print()` function can be used to output a string to the standard output (screen by default).
- Interpreter switches are either specified on the command line or in the script itself as a she-bang line for Unix systems.
- Comments are preceded by the # symbol and can be placed anywhere in your program.
- A Perl program consists of statements, with each statement terminating with a semicolon.
- Integers can be expressed in hexadecimal, octal or decimal notation.
- Double-quoted string literals perform variable substitution, and recognize a number of escape characters.
- Single-quoted string literals do not perform variable substitution.
- Scalar data represents a single piece of data. It can be a literal or a scalar variable.
- List data represents a set of scalar data. Arrays and hashes belong to this type.
- An array uses a zero-based subscript to refer to the element being referred to, while hash elements are identified by the key.

Chapter 3

Manipulation of Data Structures

3.1 Scalar Variables

I have qualitatively described how a scalar variable looks like in the previous chapter. Now we are going to look at how you can use it in your program.

You refer to a variable by appending the identifier of the variable to the symbol $. For example, a variable named LuckyNumber is written as `$LuckyNumber`.

3.1.1 Assignment

In Perl, you are not required to declare a variable before it is being used. However, before you ever use a variable in your program you should give it a default value. To assign a value to the variable we use the **assignment operator** (=). For example, we can assign the value 18 to the variable `$LuckyNumber` as follows:

```
$LuckyNumber = 18;
```

This statement is interpreted as follows: the value on the right hand side of the assignment operator is assigned to the data structure on the left. In this case, 18 is assigned to `$LuckyNumber`.

However, the right hand side of the assignment operator is not confined to a literal only. The right hand side of an assignment operator is actually treated as an **expression**. An expression consists of a sequence of operations which evaluate to a value by means of operators. For example, `(6 + 5) * 2` is an expression consisting of two operations (* denotes multiplication). Evaluating an expression means to deduce the result of the expression, by evaluating each operand, and applying the operators in a certain order (subject to operator precedence and associativity) to transform the expression into the value. The expression in this example evaluates to the scalar value 22. You will learn more about operators in the next chapter. Therefore, if we have another variable `$Num` which has the value of 8, executing the statement `$LuckyNumber = $Num` causes `$Num` to be evaluated on the right hand side, and thus its value, that is 8, is assigned to `$LuckyNumber`. So this is essentially `$LuckyNumber` = 8.

Cascaded assignment is also allowed, e.g. `$a = $b = 8;` First, 8 is assigned to `$b`, and then the value of `$b` is assigned to `$a`. The net effect is that the two variables both have the value 8.

In some other programming languages, to supply a default value before you first use a variable (**initialization**) is very important. For C/C++, you need to declare a variable before it is used. This reserves memory space for this variable. However, it is not required in C/C++ that a value needs to be assigned during variable declaration. A variable in this state is described as **undefined**. Getting a value of an undefined variable poses a very subtle source of error (some garbage values are returned — that is, arbitrary value without any meaning) that yields surprising results, and is very difficult to debug. In Perl, if you use a variable that is not initialized (for example, printing its value), the value **undef** (undefined) is returned. This is a special value that gives different values in different contexts. Contexts will be introduced in the last section of this chapter and, simply put, it is 0 in numeric context (i.e. if a number is expected), an empty string in string context (i.e. a string is expected) or FALSE in "Boolean" context (i.e. when either TRUE or FALSE is expected). However, if you have specified the `-w` switch, the interpreter should have warned you on using uninitialized variables. The use of uninitialized variables is not a good practice, and you have to ensure that all variables are given a value before being used.

As already discussed, you can store scalar data of different types in a variable during the lifetime of the script, so you can now assign the string literal "eight" to `$LuckyNumber`. Perl will just happily accept it. Of course, you should try to avoid it, as described in the previous chapter.

NOTES

You may see the terms **lvalue** and **rvalue** in some other books or documentation. An lvalue refers to any valid entities that can be placed on the left hand side of the assignment operator, while an rvalue refers to any valid entities that can be placed on the right hand side of the assignment operator. A list, an array or scalar variable can be an lvalue. Literals (scalar in sense) are not lvalues. Different programming languages have different lvalues and rvalues.

For example, these are lvalues (of course they can be rvalues as well):

```
 $var
($var, @lst)
```

These are not:

```
 46
("mystuff", "apple")
```

3.1.2 Nomenclature

Before you proceed, I will first talk about how to choose an identifier for a variable (and arrays or hashes alike). An identifier should start with a letter (A-Z, a-z) or underscore (_). Subsequent letters may be alphanumeric characters (A-Z, a-z, 0-9) or an underscore (_). No spaces are allowed in the middle of an identifier. There is one more important point. *Perl is at all times case-sensitive*. That means it differentiates lowercase and uppercase characters. The `print()` function you saw in the Hello World example cannot be replaced by Print, PRINT or anything else. Similarly, `$var` and `$Var` are two different variables. The last point to note is that identifiers cannot be longer than 255 characters (long identifiers are time-consuming to type and difficult to interpret — please avoid them).

NOTES

If you read on, you will discover that Perl has many builtin predefined variables which do not follow such a nomenclature scheme. For example, $1 - $9 are reserved for backreferencing in pattern matching (see Chapter 9 "Regular Expressions"), or other more awkward looking variables like $,, $", $_ ... just to name a few. Because devoting a whole chapter just to introduce them would be too boring, I will introduce them as needed throughout the text. Alternatively, the perlvar manpage describes these Perl predefined variables in detail.

Another point to note is that the name of a variable, array or hash is formed by a symbol ($ for variable) and the identifier. Therefore, $Var, @Var and %Var can coexist. Although they have the same identifier, they are still unique names because the symbols are different. Also, you may use "reserved words" for the identifier, e.g. $print because the symbol before the identifier tells Perl that this is a variable. There is thus no ambiguities.

3.1.3 Variable Substitution

It's time to talk about **variable substitution**. I told you that single-quoted strings do not allow variable substitution, while double-quoted strings can. Variable substitution means that variables embedded in a double-quoted string will be substituted by their values at the time the statement is evaluated. Consider this example:

EXAMPLE 3.1 Celsius → Fahrenheit Converter

```
1  #!/usr/bin/perl -w
2
3  # c2f.pl
4  # A Celsius->Fahrenheit Converter
5
6  # Print the prompt
7  print "Please enter a Celsius degree > ";
8  # Chop off the trailing newline character
9  chomp($cel = <STDIN>);
10
11 $fah = ($cel * 1.8) + 32;
12
13 # print value using variable interpolation
14 print "The Fahrenheit equivalent of $cel degrees Celsius is $fah\n";
```

Line 9 looks awkward, but here we perform two operations, to accept user input and remove the trailing newline character. You don't have to concern much about this statement yet, as this will be described in Chapter 8.

Line 11 calculates the Fahrenheit temperature, and when execution of the script reaches line 14, it sees the two variables $cel and $fah. Then Perl replaces them with the values the two variables carry at that instant, and the resulting string is output to the screen.

There is one problem arising from variable substitution. What if we have some words immediately following the variable, without even a space? Perl has provided a nice facility to get around this situation. You may put a pair of curly brackets around the identifier name to separate it from the surrounding text, e.g.

```
print "${Num}th Edition";
```

3.1.4 substr() — Extraction of Substrings

Frequently you have to extract a sequence of characters from a string. In other words, you extract a substring from it. Perl provides you with the substr() function to extract a substring, provided that you already know in advance where to start extracting it.

The syntax of the substr() function is as follows:

```
substr STRING, OFFSET
substr STRING, OFFSET, LENGTH
substr STRING, OFFSET, LENGTH, REPLACEMENT
```

As you can see, substr() can take 2-4 parameters depending on your needs. STRING is the string from which extraction is performed. OFFSET is a zero-based offset which indicates the position from which to start extraction. The first character of any string has an OFFSET 0, and 1 for the second character etc. In Perl, OFFSET can be negative, which counts from the end. For example, the last character of the string can be represented by the OFFSET -1. LENGTH is the number of characters to extract. If it is not specified, it extracts till the end of the string. The extracted substring is returned upon evaluation. Here are some examples:

```
$string = "This is test.";
print substr($string, 5);       # is test.
print substr($string, 5, 2);    # is
```

If REPLACEMENT is specified, the substring is replaced by the string obtained by evaluating REPLACEMENT, and the substring being replaced is returned. Alternatively, you may put substr() on the left hand side of an assignment operator, and REPLACEMENT on the right. Here is an example which replaces a substring of length 0 with the replacement string "not a ", that implies inserting it at position 8:

```
substr($string, 8, 0, "not a ");        ## First method
substr($string, 8, 0) = "not a ";       ## Second method
print $string;                  # This is not a test.
```

3.1.5 length() — Length of String

You can find out the length of a string by using the length() function. The only parameter for the length() function is the string itself. It returns the number of characters in the string.

Scalar variable is a very simple data structure. In the next section we are going to deal with lists, arrays and hashes that are a lot more interesting to play with.

3.2 Lists and Arrays

Arrays are named entities, lists are not. The relationship between a **list** and an array is very much similar to that between a literal and a scalar variable. A list is merely an ordered set of elements. An array is just like a list but with a name thus can be referenced through an array variable. Studying the behaviour of lists allow us to progress naturally into arrays and hashes in subsequent sections.

3.2.1 Creating an Array

Each item in the list is called an **element**. To create a list, simply delimit (separate) the elements with commas (,) and surround the list with a pair of parentheses. For example a list containing the names of some colours can be written as

```
("red", "orange", "green", "blue")
```

An array can be created by assigning a list to an array variable. An array variable starts with the symbol @ (compare with the case of $ for scalar variables). Therefore, an array can be set up containing the list above, e.g.

```
@colors = ("red", "orange", "green", "blue");
```

Alternatively, you may use the equivalent method of per-item assignment (to be discussed shortly):

```
$colors[0] = "red";
$colors[1] = "orange";
$colors[2] = "green";
$colors[3] = "blue";
```

This array contains 4 elements. A null array is simply (). Lists can be nested, that is, a list may contain other lists or array variables as an element. However, the embedded lists will be expanded and merged with the container list. Any null lists or null arrays are removed. For example, if

```
@unix = ("FreeBSD", "Linux");
@os = ("MacOS", ("Windows NT", "Windows ME"), @unix);
```

`@os` is expanded into

```
@os = ("MacOS", "Windows NT", "Windows ME", "FreeBSD", "Linux");
```

In the following example, `@result` will become a null array. Note that null lists are ignored.

```
@nullarray = ();
@result = ((), @nullarray);
```

A useful operator that worths mentioning here is the **range operator** (..). If you would like to generate an array of consecutive integers this operator may come in handy, as you no longer have to use a loop to do it. Example:

```
@hundrednums = (101 .. 200);
```

However, the numbers must be in ascending order. If you would like to have an array of consecutive integers in descending order, you may construct it in ascending order using the range operator, and then reverse the position of the items using the `reverse()` function:

```
@hundrednums = reverse (101 .. 200);
```

3.2.2 Adding Elements

We know from the previous section that if we place an array variable or a sublist as an element of a list, the array variable or the sublist would be expanded and merged with the parent list, and in this operation the identity of the original array variables or sublists are lost. That means, you can no longer tell from the resulting list if a particular element originates from a sublist or an array variable. Therefore, it is natural to conclude that two arrays can be merged together by this operation:

```
@CombinedArray = (@Array1, @Array2);
```

The resulting array contains all the elements in `@Array1`, followed by that of `@Array2`. To append a scalar element to the end of an array, you can write, for example,

```
@MyArray = (@MyArray, $NewElement);
```

We can also append a list of scalar data to the end of an array by using the `push()` function. The syntax of the `push()` function is

```
push ARRAY, LIST;
```

where ARRAY is the array to which the list data are to be appended. LIST is a list specifying the elements to be appended to ARRAY. The mechanism of the `push` function is not much different from the interpolation of lists above, and I am more accustomed to the previous one than using the `push` function, because it is more intuitive to understand.

`push` is a function. A function returns some values (not necessary scalar, can be list data as well for certain functions) after the operation is finished. For this function, the number of elements after element addition is returned. Consider the example below:

```
$NumElements = push(@MyArray, @list);
```

Note that I have added the parentheses around `@MyArray` and `@list`. The parentheses here are not actually necessary, but I added them here to make it obvious the parameters of the function. The return value is assigned to `$NumElements`. In the next chapter, you will learn operator precedence, which describes in detail when and where you should add parentheses. For the time being, stick to the way I have been doing and it would be fine.

On the other hand, this operation inserts the element at the beginning of the array:

```
@MyArray = ($NewElement, @MyArray);
```

`unshift` is a function that inserts a list at the beginning of an array, and returns the number of elements after the operation. The syntax is

```
unshift ARRAY, LIST;
```

where ARRAY is the array to which elements are added, and LIST is the list that is inserted at the beginning of ARRAY. Consider the following example:

EXAMPLE 3.2 Demonstrating unshift()

```
1  #!/usr/bin/perl -w
2
3  @alpha1 = ("a", "b", "c");
4  @alpha2 = ("d", "e", "f");
5
6  unshift @alpha2, @alpha1;
7  $, = " ";   # Prints a space in between elements
8  print @alpha2;
```

Again, disregard the line numbers that are for illustration only. Note that the ordering of the items of `@alpha1` is preserved in `@alpha2`. What is worth noting is on line 7. We assign a space to an odd-looking variable, but what's that? This is an example of Perl predefined variables, and `$,` is known as the **output field separator**. Without line 7, you would most likely get an output like "abcdef" because by default this variable is an empty string. Note that the `print()` function can accept a list of parameters. The output field separator is the string that is added in between list elements in a `print` operation. You may assign any valid string literals to this variable. In this example, I put a space in between elements to make them easier to read.

3.2.3 Getting the number of Elements in an Array

There are two ways in which you could obtain the number of elements stored in an array.

The first method is to employ the concept of context. By evaluating an array in scalar context, we can obtain the number of elements in an array. You may not understand what this is for the time being, but we will return to this example when we come to contexts later in this chapter. In the following example, the number of elements in `@colours` is asssigned to `$numElements`:

```
$numElements = @colours;
```

The second method is a little bit clumsy to explain. In Perl, you can find out the subscript of the last item of an array by replacing the symbol @ with `$#`. For example, the subscript of the last item in `@Array` is given by `$#Array`.

For historical reasons, Perl provides a facility for users to specify the subscript of the first element of an array. This is specified by assigning an integer to the predefined variable `$[`. This is 0 by default, and that's why I said subscripts start from 0. However, some people may be accustomed to using 1 as the starting index (especially those who have used some "antique" programming languages).

This value is not necessarily 0 or 1. You may set other values as well. However, you cannot assign values to this variable more than once. Although Perl provides this facility, you are advised not to use it because of potential confusions that may arise, especially if your project consists of a number of files.

With both the start index and the index of the last element, we can get the number of elements by using the formula:

Number of elements = Last Index - Starting Index + 1
= $#Array - $[+ 1

As the start index is 0 by default (that is, if you don't specify otherwise), you may assume that the number of elements of @Array is given by

```
$#Array + 1
```

3.2.4 Accessing Elements in an Array

After we have added items to an array or list, we can access any of its elements by using the **subscript operator** ([]). For example, if we would like to retrieve the third element (remember that subscripts count from 0) from @colours and return the value to $col3, we can write

```
$col3 = $colours[2];
```

Note that the symbol is $ instead of @ on the right hand side of the assignment operator. In Perl, because the value returned is a scalar value, the symbol $ is used. Though looking awkward, you can definitely use the subscript operator on a list, like this:

```
$col3 = ("red", "orange", "green", "blue")[2];
```

Obviously, using the subscript operator in this way is not really useful.

What about specifying a negative subscript? Negative subscripts count backwards, from the last element of the array. The last element has the subscript -1, and the next-to-last element has the subscript -2. Therefore, the first element of the array @array has the subscript -@colours (due to scalar context). Normally, negative subscripts are used as a convenience method to retrieve the last element of an array. At least, @array[-1] is definitely easier to understand than @array[$#array].

An **array slice** is a subset of elements from all the elements in an array. The subscript operator is not confined to one subscript only. You may specify a list of subscripts using the comma operator (,) and the range operator (..). You use the range operator to specify subscripts (must be integral) in a given range, and the comma operator to specify each subscript individually. Nothing will stop you from using both operators together, as in the following example:

EXAMPLE 3.3 Array Slices

```
1 #!/usr/bin/perl -w
2
3 @alpha = ('a' .. 'z');
```

```
4  @slice = @alpha[4, 10 .. 15];
5
6  $, = " ";
7  print @slice;
```

In this example, the array slice `@slice` contains the 5th element, and 6 elements starting from the 11th element of `@alpha`. The resulting output is thus "e k l m n o p". Note that because the array slice contains a list of values, the symbol of `@alpha` is @ on line 4.

We can, of course, modify the value of any element in an array. To do this, just assign a scalar value to the corresponding array element. Similar is the case for slice assignment, in which a list of scalar data is assigned as a list. For example:

```
$colours[2] = "violet";

# The followings are identical
@colours[2,4] = ("violet", "blue");
$colours[2] = "violet"; $colours[4] = "blue";
@colours[4,2] = ("blue", "violet");
```

If you specify a subscript that is larger than `$#array`, the size of the array shall grow in order to accommodate the newly added element. In this example, because `@nums` contains originally 3 elements, assigning a value to the 5th element leaves a "gap" at the 4th element, not assigned any values. The value of this element is `undef`.

EXAMPLE 3.4 Array Expansion

```
1  #!/usr/bin/perl -w
2
3  @nums = (3, 4, 5);
4  $nums[4] = 7;   # (3, 4, 5, undef, 7) expected
5
6  $, = "\n";
7  print @nums;
```

To show that there is an empty gap at the 4th element, the output list separator is set to \n. So you would see a line like "Use of uninitialized value at eg0304.pl line 6" between 5 and 7 in the output if warnings are enabled. That's because retrieving the value of an uninitialized value produces this Perl warning.

3.2.5 Removing Elements

We can use the `pop` function to remove the last element of an array. It also returns the value of the item being removed. Syntax:

```
pop ARRAY;
```

In this example, the last item of `@MyArray` is removed and its value is assigned to `$retval`:

```
$retval = pop(@MyArray);
```

On the other hand, the `shift` function removes the first element of the array, so that the size of the array is reduced by 1 and the element immediately after the item being removed becomes the first element of the array. It also returns the value of the item being removed. Syntax:

```
shift ARRAY;
```

If ARRAY is empty, `undef` is returned.

3.2.6 `splice()`: the Versatile Function

There is a generalized function for adding and removing elements from an array. The `splice` function is so general that it can do what `push`, `pop`, `shift` and `unshift` does. The syntax is:

```
splice ARRAY, OFFSET [, LENGTH [, LIST]];
```

In this tutorial, the parts in slanted font denote optional parameters. These parameters are optional in the sense that in some situations they are optional; but not in other situations. I use [] to label that a parameter is optional. Note that [LIST] is placed within another optional parameter LENGTH. This means that if you have to specify the parameter LIST, you must also specify LENGTH; but not vice versa. If you specify the parameter LENGTH, you may or may not supply the parameter LIST.

In general, this function removes LENGTH elements starting from the element of subscript OFFSET of ARRAY, and inserts LIST at OFFSET if any. Simply put, the list `@ARRAY[OFFSET .. OFFSET + LENGTH - 1]` is replaced by LIST. The syntax shows that this function takes three forms, and I am going to describe them one by one.

```
splice ARRAY, OFFSET, LENGTH, LIST
```

This performs exactly the action above. In the example below, an array containing the 26 lowercase alphabets was built, and 5 elements starting from the 5th element (i.e. the letter "e") is converted to uppercase. This is done on line 2. First `@alpha[4 .. 8]` contains the 5 letters that are to be converted to uppercase (remember that subscripts start at 0). The `map` function calls the `uc` function (uppercase) for every element in this list, thus converting `("e", "f", "g", "h", "i")` to `("E", "F", "G", "H", "I")`. The `splice` function, therefore, replaces the lowercase list with the uppercase one. `uc` and `map` would be covered later.

EXAMPLE 3.5 splice

```
1 #!/usr/bin/perl -w
2 # Example 3.5 - splice
3
4 @alpha = ('a' .. 'z');
5 splice @alpha, 4, 5, map(uc, @alpha[4 .. 8]);
6 $, = ' ';
7 print @alpha;
```

What you will see on the screen should be all lowercase alphabets except E, F, G, H and I.

```
splice ARRAY, OFFSET, LENGTH
```

If you don't specify the replacement list, the action is merely removing LENGTH elements from ARRAY starting from subscript OFFSET.

```
splice ARRAY, OFFSET
```

If you don't specify the LENGTH, Perl assumes that all elements starting from OFFSET are removed. OFFSET is just a subscript and can be negative as well. A negative value specifies the OFFSET is counted from the end of the array as mentioned earlier, e.g. -1 means the last item, -2 means the second last item etc. Therefore, `pop @MyArray` can be equivalently accomplished by `splice @MyArray, -1`.

The following table summarizes how you can use `splice` in place of other functions discussed earlier (still remember that in Perl you always have a number of ways to do the same task?). You may find the equivalent method for `push()` strange, but after you have learnt the whole theory behind contexts you would understand it. We would use this illustration again when we come to contexts.

Function	Equivalent Method
`push(@Array, $x, $y)`	`splice(@Array, @Array, 0, $x, $y)`
`pop(@Array)`	`splice(@Array, -1)`
`shift(@Array)`	`splice(@Array, 0, 1)`
`unshift(@Array, @x)`	`splice(@Array, 0, 0, @x)`
`$Array[$x] = $y`	`splice(@Array, $x, 1, $y)`

Table 3.1: Relationship of some array functions with splice()

3.2.7 Miscellaneous List-Related Functions

There are a number of useful functions that allow you to manipulate list data in Perl.

```
join STRING, LIST
```

The `join()` function concatenates a list of scalars into a single string. It takes a `STRING` as its first argument which is the separator to be put in between the list elements `LIST`. For example,

```
    join '+', 'apple', ('orange', 'banana')
```

evaluates to

```
    apple+orange+banana
```

```
reverse LIST
```

In a list context, the reverse() function returns a list whose elements are identical to that of LIST except the order is reversed. For example:

```
print join " ", reverse 'a'..'e';   # e d c b a
```

The map() function takes on one of the following two forms:

```
map BLOCK LIST
map EXPR, LIST
```

The map() function iterates over every item in the LIST, sets $_ to the item concerned and executes BLOCK or EXPR on each iteration. The return value is a list consisting of the result of evaluation of all iterations. BLOCK is a code block enclosing a sequence of statements to be executed. EXPR can be any expression. Consider this example:

```
@names = ('ALICE', 'tOm', 'JaSON', 'peter');
print join(', ', map { ucfirst(lc($_)); } @names), "\n";
```

The output is

```
Alice, Tom, Jason, Peter
```

This example prints out each of the names in @names so that they all start with capital letters while the other characters are in lowercase. This is accomplished by first converting all characters to lowercase by the lc() function, and the first letter is capitalized using the ucfirst() function. This process is performed for each name in @names. The same expression may be rewritten as

```
@names = ('ALICE', 'tOm', 'JaSON', 'peter');
print join(', ', map (ucfirst(lc($_)), @names)), "\n";
```

The first form of map() is generally preferred to the second one because it is more flexible.

The sort() function can be used to sort a list. By default, the sort() function sorts lexicographically. The items are ordered by comparing the items stringwise (using the cmp operator, the specifics of which will be introduced in the next chapter). This comparison is case-sensitive, because it is based on the ASCII values of each character. The sorted list is returned by the sort() function, while the original list remains intact.

```
sort ('bear', 'Post', 'ant');   # ('Post', 'ant', 'bear')
```

Note that while comparing stringwise, capital letters are considered "smaller" than lowercase letters.

To make the sort routine generic and allow sorting in any arbitary order, you may override the default sort criteria with your own rules. The method is to insert a code block before the list, similar to the case for map(). The value resulted from evaluation of the block determines how the items are sorted.

The principle of constructing the contents of the block is too advanced at this stage. The rest of the details can be found in the next chapter. In the following I list several most commonly used sort criteria that you are likely to find useful:

```
sort { $a <=> $b } @list;   # ascending numerical order
sort { $b <=> $a } @list;   # descending numerical order
sort { $a cmp $b } @list;   # ascending lexicographical order (default)
sort { $b cmp $a } @list;   # descending lexicographical order

# case-insensitive ascending lexicographical order
sort { lc($a) cmp lc($b) } @list;       # or use uc()
```

3.2.8 Check for Existence of Elements in an Array (Avoid!)

Many new programmers, and programmers who have programmed in other languages tend to use arrays so extensively that are sometimes inappropriate from performance considerations. One classical problem is to deduce if a particular piece of scalar data matches any of the elements stored in an array (in alternative terminology, check for a **hit** or a **miss**). In some languages like BASIC where a hash is not a builtin type, there is still little excuse to solve this problem by searching an array, although you can still implement a hash yourself (For those who are interested in implementing a hash themselves, please read Appendix A).

In general, you can check if a certain element exists in an array by **linear search**. That is, you search from the first element up to the end of the array. In the following code snippet, a random list containing 100 entries is generated randomly and you can enter a number to be searched. The program then searches for the number. When the search ends, it returns whether it is found, and how many elements the program has searched.

EXAMPLE 3.6 Linear Search of an Array

```
1  #!/usr/bin/perl -w
2
3  # Linear search of an array
4
5  # Note that if you later on want to search for something from a
6  # list of values, you shouldn't have used an array in the first
7  # place.
8
9  # Generating 100 integers
10 $NUM = 100;
11 $MAXINT = 5000;     # 1 + the maximum integer generated
12
13 srand();        # initialize the randomize seed
14
15 print "Numbers Generated:\n(";
16 for $i (1 .. $NUM) {
17     push @array, sprintf("%d", rand(1) * $MAXINT);
18     print $array[$i-1];
19     print ", " unless ($i == $NUM);
20 }
```

```
print ")\n\n";

print "Please enter the number to search for >> ";
chomp($toSearch = <STDIN>);

# Linear search here
$counter = 0; $hit = 0;
foreach $num (@array) {
    $counter++;
    if ($num == $toSearch) {
        print "\"$toSearch\" found at subscript ", $counter - 1, "\n";
        $hit = 1;
        last;
    }
}
if ($hit == 0) { print "\"$toSearch\" not found in array.\n"; }
print "Number of comparisons: $counter/", scalar(@array), "\n";
```

The code itself is not very important here. In fact, many new constructs used in this program have not been discussed yet. The intent is for you to run the program instead of reading the source code (but you may do it). Try to look at the list of values printed, and try to enter a number that is not in the array, a number that appears early in the list, and a number that appears around the end of the list. A fraction is printed on the screen. The one on the left is the number of comparisons performed, while the one on the right is the number of elements in the array (which should be 100 in this example). You may also want to increase `$NUM` to increase the number of integers generated. Of course, in this case you will need to increase `$MAXINT` accordingly to minimize the chance of duplication of values.

Note that it is possible to have duplicate values in the array owing to the random nature of the random number generator. In this program, the first occurrence (i.e. with the lowest subscript) will be returned.

An alternative scheme is to use **binary search**, a rather classical topic in elementary computer science texts on algorithms. By using binary search, the whole set of values needn't be searched in the worst case. In the worst case only $log_2 n$ comparisons are required. However, there is an important requirement — the list needs to be already sorted. If not, sorting needs to be performed first. Here is the above example rewritten that employs the binary search algorithm:

EXAMPLE 3.7 Binary Search of an Array

```
#!/usr/bin/perl -w

# Binary search of an array

# Note that if you later on want to search for something from a
# list of values, you shouldn't have used an array in the first
# place.

# Generating 100 integers
$NUM = 100;
$MAXINT = 5000;     # 1 + the maximum integer generated
```

```
12
13  srand();        # initialize the randomize seed
14
15  print "Numbers Generated:\n(";
16  for $i (1 .. $NUM) {
17      push @array, sprintf("%d", rand(1) * $MAXINT);
18      print $array[$i-1];
19      print ", " unless ($i == $NUM);
20  }
21  print ")\n\n";
22
23  print "Please enter the number to search for >> ";
24  chomp($toSearch = <STDIN>);
25
26  # First sort it in ascending numerical order
27  @sortedArray = sort {$a <=> $b} @array;
28
29  # Binary search here
30  $counter = 0;
31  $start = 0; $end = $#sortedArray; $mid = 0; $hit = 0;
32  $mid = sprintf("%d", ($start + $end)/2);
33  while ($start <= $end) {
34      $counter++;
35      print "Searched: ", $sortedArray[$mid];     # which element is being ¶
            searched
36      print " in #[$start, $end] [";      # the subscript range
37      print $sortedArray[$start], ", ", $sortedArray[$end], "] \n";
38      if ($sortedArray[$mid] == $toSearch) {
39          # a hit!
40          print "\n\"$toSearch\" found!\n";
41          $hit = 1;
42          last;
43      } elsif ($sortedArray[$mid] > $toSearch) {
44          # decrease upper boundary -> mid value
45          $end = $mid - 1;
46      } else {
47          # update lower boundary -> mid value
48          $start = $mid + 1;
49      }
50      $mid = sprintf("%d", ($start + $end)/2);
51  }
52
53  if ($hit == 0) { print "\n\"$toSearch\" not found in array.\n"; }
54  print "Number of comparisons: ", $counter, "/", scalar(@sortedArray), "\n";
```

This implementation is even more complicated than that of the linear search. What is worth noting is the very low number of comparisons required in any cases (For 100 integers the fraction printed in any case should be at most 7/100 [1]). That means among the 100 integers in the list, at most 7 elements searched is sufficient to deduce whether any specified number exists in the array. In

[1] The maximum number of comparisons can be calculated by $ceiling(log_2(n))$ where $ceiling(y)$ is the minimum integer that is greater than or equal to y. In this case $ceiling(log_2(100)) = 7$.

Chapter 5, as an exercise, you would be asked to convert this program into the recursive form.

Binary search works as follows. The list is sorted in either ascending or descending order (assume we have the list sorted in ascending order here). In this algorithm we always maintain two important variables — `$start` and `$end`. Initially they are set to 0 and 99 (the subscript of last element) respectively. Then the element in the middle (subscript 44) is examined and compared with the number to be searched. If the number examined is larger than the input number, that means the input number, if exists, would have a subscript less than 44, so we can shrink the range by reducing `$end` to 43. Otherwise it would have a subscript greater than 44, so we increase `$start` to 45. Such a bisection process is repeated with this new set of `$start` and `$end`, until the number is found or `$end` is less than `$start`. Note that on each iteration the range examined is halved, and that's why the maximum number of comparisons is $log_2(n)$.

An example of binary sort is shown in Figure 3.1. 18 is to be searched and is found on the third iteration. The red boxes denote the element in the middle of the range being searched, and those in grey are those that are bypassed.

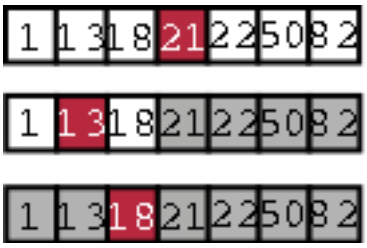

Figure 3.1: An illustration of binary sort

Although it seems binary search performs a lot better compared with linear search, there is an important catch here — most arrays we encounter are not sorted, so the cost of sorting cannot be simply ignored in practice.

As you can possibly see in both cases when `$NUM` is large, neither approach is performing in an efficient manner, especially when misses occur because in the linear search case, every element in the array has to be searched in the worst case (when a miss occurs); while in the binary search case, the array needs to be sorted first, and the average sorting time of an array grows with the array size [2], so combining sorting and searching may be even slower than the linear search algorithm. Towards the end of the next section we shall redo this code with a hash, and you ought to find the code can be made cleaner and a hit or a miss can be determined faster in general, especially with large array sizes.

3.3 Hashes

Hash is a special kind of data structure. There are several characteristics associated with it. It practically takes very short time to deduce whether any specified data exists. Also, the time it takes does not largely depend on the number of items stored. This is important because hashes are usually used for applications that handle a large amount of data.

[2] In algorithmic analysis, the best sort algorithms can attain the time complexity of $O(nlogn)$. That is, the sorting time t is in the form $t = Knlog(n)$ where K is a constant and n is the size of the array. The function grows even faster than n, so performance degrades would be evident at relatively large values of n.

An array is simply a contiguous block of memory and is nothing more than that. In order to support the characteristic stated above, hashes require a slightly more complicated internal structure. This is outlined in Appendix A for your reference. It explains the general principles that further Perl knowledge is not necessary in order to understand it. However, in this section we are dealing with how we use hashes in Perl, and will not discuss the peculiarities of them here.

As a quick review, each item in a hash has a key and a value. The key, which is a string, uniquely identifies an item in the hash. The value is any form of scalar data. Hash variables start with the symbol %.

3.3.1 Assignment

We may assign a list to a hash variable. In this case, the list will be broken up two-by-two, the first one as the key and the second as the value:

```
%Age = ('Tom', 26, 'Peter', 51, 'Jones', 23);
```

Because a hash contains multiple key-value pairs, this alternative syntax may seem more intuitive to look at:

```
%Age = ('Tom' => 26, 'Peter' => 51, 'Jones' => 23);
```

The symbol => is defined as an almost equivalent symbol to the comma, so in general anywhere a comma is needed you can replace it with this symbol. However, the use of => is particularly intuitive for data that occur in pairs. In addition, Perl allows even less keystrokes by omitting the quotation marks:

```
%Age = (Tom => 26, Peter => 51, Jones => 23);
```

This is because Perl always interprets the word before the symbol => as a double-quoted string. However, if you omit the quotes, the key cannot have embedded whitespace (space, tabs and so on). To specify a key with embedded whitespace, the quotes must be specified. Also, because Perl uses the semicolon, not a newline, to mark the termination of a statement, you can make the hash assignment better to look at by writing it in multiline form:

```
%Age = (
    Tom => 26,
    Peter => 51,
    Jones => 23,
);
```

Note that on the final line there seems to be a superfluous comma there. Yes, it is. The final comma before the closing parentheses can be omitted. But it is customary to have it there in the above multiline form because it is likely that you may append more key-value pairs later on. Then you will have to add the comma back, and it will result in an error if you have omitted to do so. Bear in mind that the four formats above are identical, and you may choose the form which looks best to you. An empty hash is an empty list assigned to a hash variable, similar to the case of arrays.

NOTES

One note about why we can omit the quotes here. According to the perldata manpage, a word that has no other interpretation in the grammar will be treated as if it were a quoted string. For example, if an unquoted word is not a reserved word, filehandles, labels etc. Perl will automatically treat it as a bareword. However, avoid barewords consisting entirely of lowercase letters, because all Perl reserved words are in lowercase. As Perl is case-sensitive, this eliminates the possibility of potential name clashes in future versions of Perl.

Because in the assignment operation Perl expects a list as the rvalue, apart from lists you may as well assign an array, or even a subroutine returning a list to the hash variable. Note that the hash variable provides a list context here.

If we assign a hash variable to an array, or anywhere a list is expected, the key-value pairs stored in the hash will be returned in list form. However, because of the way the key-value pairs are stored in a hash, they may be (actually most likely) returned in an order different from when they were put into the hash.

3.3.2 Accessing elements in the Hash

Accessing an element in a hash is similar to that from an array, except we replace square brackets with curly ones and instead of an index, the key is used. Here's an example:

```
print $Age{Tom};
```

As you will learn in Chapter 10, a word appearing inside the curly braces not enclosed by quotes is evaluated as a string. Therefore, Tom is equivalent to 'Tom' in this case. To associate a scalar value to a key is as simple as:

```
$Hash{'Key'} = $value;
```

If Key already exists, it is assigned the supplied value; otherwise, a new key-value pair is added to the hash.

3.3.3 Removing Elements from a Hash

Perl provides the `delete` function for removing a specified key-value pair from a hash. Here's an example:

```
delete $Age{Tom};
```

This function returns the deleted scalar value associated with the specified key(s). So if you `print` the return value in the above statement, the value(s) deleted would be displayed on screen.

To delete all key-value pairs in a hash, you can of course use a loop to do it, but this is slow and it would be more efficient to use either method below:

```
%Age = ();
```

or

```
undef %Age;
```

Here is a special example for `delete` that is worth mentioning:

```
1  %Age = (
2      Tom => 26,
3      Peter => 51,
4      Jones => 23,
5  );
6  @temp = delete @Age{'Tom', 'Peter'};
7  $, = " ";
8  print "Deleted values:", @temp, "\n";
9  print "Remaining keys:", keys @Age;
```

On line 6, we delete multiple key-value pairs from `%Age`. Note that the function returns a list of values associated with the deleted keys, so the symbol used should be `@`.

Line 9 uses the `keys()` function. It returns a list of keys in the specified hash. There is a corresponding `values()` function that returns a list of values contained in the hash. Recall the `sort()` function mentioned earlier. Apart from arrays, you may use the `sort()` function together with the `keys` and `values` function to specify how to order the hash items. For example,

```
%array = (
    '3' => 'apple',
    '11' => 'orange',
    '5' => 'banana',
);
@key = sort { $a <=> $b } keys %array;      # ('3', '5', '11')
@value = sort { $a cmp $b } values %array;      # ('apple', 'banana', 'orange')
```

3.3.4 Searching for an Element in a Hash

You may test if a particular key exists in a hash by using the `exists` function. However, even if the key exists in the hash, the value associated may be undefined. Use the `defined` function to test if the value is defined.

Recall earlier we had a program that generated 100 integers in random and you look for the existence of a particular number in the list. We now present the version using a hash. The number of comparisons taken is not shown, as you cannot get this information with a builtin hash. Of course if you build a hash yourself you would be able to see the efficiency of hashes. At least, you will find the implementation is a lot easier compared with the two previous approaches.

EXAMPLE 3.8 Searching for an Element in a hash

```
1  #!/usr/bin/perl -w
2
```

```
3  # Search for an element in a hash
4
5  # Generating 100 integers
6  $NUM = 100;
7  $MAXINT = 5000;     # 1 + the maximum integer generated
8
9  srand();        # initialize the randomize seed
10
11 print "Numbers Generated:\n(";
12 for $i (1 .. $NUM) {
13     $valueToInsert = sprintf("%d", rand(1) * $MAXINT);
14     $hash{$valueToInsert} = 0;  # in fact, any values can be assigned here
15     print $valueToInsert;
16     print ", " unless ($i == $NUM);
17 }
18 print ")\n\n";
19
20 print "Please enter the number to search for >> ";
21 chomp($toSearch = <STDIN>);
22
23 # Hash search here
24 if (exists($hash{$toSearch})) {
25     print "\"$toSearch\" found!\n";
26 } else {
27     print "\"$toSearch\" not found!\n";
28 }
```

Notice how clean it is to determine whether the number exists in the hash in this case (line 24). In this example, the numbers are stored as keys in the hash. We use the `exists()` function to check if the key exists in the hash. This function returns TRUE if the specified key exists. Note that you can as well use `defined()` in place of `exists()` in this example. This function returns true only if the key exists in the hash, and the value is not undefined (i.e. `undef`).

Table 3.2 summarizes the difference between `defined()` and `exists()`:

Function	Key Exists	Value Undefined	Return Value
`exists()`	Yes	No	TRUE
	Yes	Yes	TRUE
	No	N/A	FALSE
`defined()`	Yes	No	TRUE
	Yes	Yes	FALSE
	No	N/A	FALSE

Table 3.2: Differences between exists() and defined()

In the example, because 0 (or anything except `undef`) is assigned as value to each key put into the hash, therefore, `exists()` and `defined()` yields the same results.

Owing to the nature of a hash, duplicate keys are not allowed. However, as I have previously noted it is possible that by using such a random generation scheme duplicate elements may be generated, and in this case exactly only 1 instance is stored, so the number of elements in the hash in this

example can be less than 100.

3.4 Contexts

The idea of contexts in Perl may appear a bit odd at first glance, but you would soon discover that contexts are actually quite intuitive to understand, because there are many real life examples resembling contexts in Perl.

First, look at these two phrases. Note that the same word "press" appears in both sentences, but their meanings in the phrases are entirely different.

Press the button
Freedom of **press**

The word "press" plays different roles in the two sentences, one acting as a verb and the other as a noun. Therefore, to deduce the meaning of the word in the phrase, we have to look at the words surrounding it, in other words, the **context**.

I'm sure you would find many other examples that exemplify how contexts play their roles in our daily lives, so I am not going to spend too much time mentioning things you may have already known. Recall that earlier in this chapter I obtained the number of elements of an array by using an assignment like this:

```
$numElements = @colours;
```

You may find this assignment rather peculiar. It seems that I have been assigning list data to scalar data, and they actually don't match! How can a scalar data store an array?

Yes. That's impossible (unless by using references - that we will see at a later part of this tutorial). Also recall that I mentioned earlier that I was "evaluating an array in scalar context". Now I am telling you what this phrase means.

Because you are now assigning something to a scalar variable, the data type expected on the right hand side of the assignment operator is naturally a scalar value. In this way, we have created a scalar context around the array `@colours`. Because a scalar value is requested instead of an array, Perl defines that by evaluating an array in scalar context, the number of elements stored in the array is returned and the value of which is assigned to `$numElements`.

It is important to bear in mind that the rules as to how a data type is evaluated in another context depend on the definition, and there is no general rules of inference. Later on, when you are to write subroutines, you will be taught to use the `wantarray()` function to determine the contexts so that you can specify what to return from your subroutine in different contexts.

Remember I have mentioned Boolean context earlier in this chapter? Actually there is not a "Boolean" context, because Perl does not have an intrinsic Boolean data type. Perl achieves this by using scalar context instead. Perl defines that the numeric zero (0), the empty string ("") and the undefined value (`undef`) are interpreted as FALSE, and all other scalar values are interpreted as TRUE. Therefore, anywhere a Boolean test is expected you can place any scalar value there instead,

and Perl shall evaluate it according to this rule.

Sometimes you may want to evaluate a list in scalar context, but it is in list context instead. A useful function about contexts is `scalar`, which provides the necessary scalar context, as in the following example:

```
print scalar(@array);
```

Without the `scalar` function, the array is evaluated in list context and so the contents of the array will be `printed`. However, if we would like to `print` the number of elements in the array instead, the `scalar` function provides the necessary scalar context for this purpose.

I do not aim at telling you everything about how different contexts give rise to different behaviours when we deal with operators at the moment. Instead, it is important to realize that operators can exhibit different behaviours depending on the context they are in. You are going to have a boring day probably reading the next chapter, as you are going to know in more details how each operator is affected by contexts.

3.5 Miscellaneous Issues with Lists

Before we start a new chapter, let us pay attention to several issues concerning lists which have not yet been mentioned above.

You have seen in the previous section that, because of evaluation of an array variable in scalar context, the number of elements in the array is returned:

```
$a = @array;
```

but things get entirely different when you write it this way:

```
$a = (35, 48, 56);
```

This is a special case, because *in scalar context* the list on the right of the assignment operator is actually a list of values delimited by the **comma operator** (,). This operator is borrowed from C/C++ and has nothing to do with lists or arrays. The behaviour of this operator is, evaluate each of the expressions (in this case, numbers) delimited by commas, and return the value resulting from the last expression. Therefore, in this example 56 is returned and assigned to `$a`. This applies to void context as well. While

```
@array = (35, 48, 56);
```

the commas there act as list argument separators. As a rule, always bear in mind that a list is only a list in list context. In other contexts the rules of comma operators apply.

Thanks to the flexible syntax of Perl, we can assign multiple values concurrently, like this (this is called list assignment):

```
($a, $b) = (11, 22);
```

Both sides are lists of the same size, so this is just a mapping: 11 is assigned to $a and 22 to $b. What about if the list on the right has more elements than the one on the left? Then the extra elements are simply ignored. What about if the list on the left has more elements? Then some variables in the left hand list shall receive the value undef, as you may expect. To swap the values of two variables, an operation that requires an additional temporary variable in most other programming languages, you can perform it in Perl simply by:

```
($x, $y) = ($y, $x);
```

Consider this example and try to guess what happens:

```
($a, @b) = (11, 22, 33, 44);
```

11 is assigned to $a. The remaining three elements are assigned to @b. But what if you do this?

```
(@a, $b) = (11, 22, 33, 44);
```

In this case, @a gobbles up all the values in the list, leaving the value undef to $b. We describe this behaviour as "greedy".

In this chapter, you have learnt how to use the three fundamental data structures in Perl, namely variables, arrays and hashes. I have also tried to give you some ideas on the concept of contexts, which is of fundamental importance in Perl. In the next chapter, I will introduce to you the operators available in Perl.

Summary

- Scalar variables are prepended with the symbol $. For array and hash variables, the symbols are @ and % respectively.
- Scalar variables can be assigned a value by using the assignment operator =. A variable holds the value undef by default.
- Variable names are case-sensitive. They should start with an alphabet or an underscore character (_). Subsequent characters may be decimal digits as well. Variables whose names not conforming to this nomenclature are predefined by Perl and serve special purposes.
- Variable substitution means variables embedded in a double-quoted string will be substituted by their respective values at the instant the string is evaluated.
- The substr() function extracts a sequence of characters from a given string.
- The length() function returns the number of characters in a given string.
- A list is an ordered set of scalar values. An array is a list associated with a name.
- An array may be created by assigning a list to an array variable.
- Nested lists are merged to form a single list when the list is evaluated, with null lists removed.
- The reverse() function returns a list whose items are identical to the input list except the items are arranged in reverse order.

- The `push()` function may be used to append a list of items to the end of an array.
- The `unshift()` function prepends a list of items to the beginning of an array.
- The subscript operator `[]` can be used to access a subset of elements in a list or an array.
- The `pop()` and `shift()` functions may be used to remove an item from an array. `pop()` removes the last item while `shift()` removes the first item.
- `splice()` is a general purpose function to add or remove array elements.
- `join()` concatenates a list of scalars into a string, inserting a string in between.
- `map()` executes a given code block for each list element, and the results evaluated are combined to form an array.
- Searching an array is not efficient. A hash should be used instead.
- A hash may be initialized by assigning it a list, whose elements are treated as a list of key-value pairs.
- To refer to a hash element, use curly braces with the key in between.
- The `delete()` function may be used to remove a key-value pair from a hash.
- You may use the `exists()` function to test if a given key exists in the hash. The `defined()` returns if a value is not `undef`.
- Behaviours of Perl operators and functions are influenced by context. The `scalar()` function forces a scalar context in an otherwise list context.
- The result of evaluating an array in scalar context is the number of items in the array. Evaluating a list in scalar or void context causes all list items to be evaluated, and the value of the last item is returned.

Chapter 4

Operators

4.1 Introduction

Operators are important elements in any programming language. They are so called because they operate on data. For those who are new to computer programming the operators with which they may be most familiar are the arithmetic operators, like addition, subtraction, multiplication and division. Yet there are many more varieties of operators in programming languages. Perl provides more operators than most programming languages I could think of, but most of them fall within one of the several categories that we would go into detail in this chapter. Although devoting an entire chapter to discuss operators is rather boring, it is important for you to understand what operators are as they act as the glue to bind pieces of data into an expression.

Arithmetic operators manipulate on numeric scalar data. Perl can evaluate an arithmetic expression, in a way similar to our daily-life mathematics.

Assignment operators are used to assign scalar or list data to a data structure. Apart from = that you learned earlier, there are other operators, like +=, that perform additional operations at the same time.

Comparison operators are used to compare two pieces of scalar data, e.g. alphabetically or numerically and returns a Boolean value. For example, you have two variables and you can use a comparison operator to deduce which one is numerically larger.

Equality operators compares two pieces of scalar data and returns if their values are identical. They may be considered special cases of comparison operators.

Bitwise operators provide programmers with the capability of performing bitwise calculations.

Logical operators can be used to do some Boolean logic calculations.

String manipulation operators manipulate on strings.

There are some other operators that do not fall into the categories above. Some of them will be covered in this chapter, and the rest would be introduced as needed in subsequent chapters in this tutorial.

Because Perl classifies all data into one of the two forms, namely scalar and list data, operators can be classified, similarly according to the number of operands, into two groups. Either the number of

operands is fixed or variable. An operator that takes on one, two and three operands are referred to as **unary**, **binary** and **ternary** operators, respectively. **List operators**, on the other hand, can take a list of arguments as operands.

4.2 Description of some Operators

In this section we shall study a number of operators. This is not intended to be a full coverage because it would be better for you to learn the rest later on in the tutorial as you accumulate more Perl knowledge. If you would like to have a detailed reference of the whole family of operators in Perl, please consult the perlop manpage.

4.2.1 Arithmetic Operators

Arithmetic operators refer to the following operators:

Operator	Description
+	Addition operator
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Modulus operator
+	Positive sign
-	Negative sign
++	Autoincrement operator
--	Autodecrement operator
**	Exponentiation operator

The operators +, -, *, / take two operands and return the sum, difference, product and quotient respectively. Note that the division operation is floating-point division. Unlike C, Perl does not offer builtin integral division. To get the integral quotient, you may use the `int()` function. For example, `int(7 / 2)` evaluates to 3.

Perl won't round a number to the *nearest* integer for you automatically. If you need this, the "correct" way to do this is

```
$num = 7 / 2;
print int($num+0.5), "\n";
```

The modulus operator is more problematic. The operands of this operand are both integers. If you feed any floating-point numbers (i.e. decimals) as operands they will be coerced to integers. Assume we are carrying out `$a % $b`.

If `$b` is positive, the value returned is `$a` minus the largest integral multiple of `$b` such that the result is still positive. For example,

`63 % 5 = 3` ($63 = 12 \times 5 + 3$, the largest multiple in this case is 12)

`-63 % 5 = 2` ($-63 = -13 \times 5 + 2$, the largest multiple in this case is -13)

If `$b` is negative, the value returned is `$a` minus the largest integral multiple of `$b` such that the result is still negative. For example,

`63 % -5 = -2` ($63 = (-13) \times (-5) + (-2)$, the largest multiple in this case is -13)

`-63 % -5 = -3` ($-63 = 12 \times (-5) + (-3)$, the largest multiple in this case is 12)

Such behaviours are so tedious that most programmers simply use the modulus operator for the first case — where both operands are positive integers, which evaluates to the remainder of `$a / $b`.

The positive/negative signs, just as in our usual mathematics, are unary operators that are affixed before a number to indicate whether it is positive or negative. Our usual convention is that the unary positive sign is not specified, because it is the default as expected.

If you know C/C++, the autoincrement and the autodecrement operators are identical to what you have learned. For those who don't know, it's worth to spend a few minutes to repeat all the details here. These operators can be placed before or after a variable. There are four possible variations:

Expression	Description
`++$var`	Prefix Increment
`$var++`	Postfix Decrement
`--$var`	Prefix Decrement
`$var--`	Postfix Decrement

The first two are autoincrement, while the remaining two are autodecrement. If autoincrement/autodecrement is performed as a statement on its own, the prefix or postfix configurations do not produce any difference. For example, both `++$a;` and `$a++;` as standalone statements increase the value of `$a` by 1. However, they are different if the operators are used as part of a statement. Consider the following examples:

A. `$b = ++$a;`
B. `$b = $a++;`

In statement A, `$a` is first incremented, and then the new value is returned. In statement B, however, the value is returned first, and then `$a` is incremented. Therefore, the value returned (and is thus assigned to `$b`) is the value before increment. The two forms differ in the order of increment/decrement and return of value. Autodecrement works in the same way, except the variable is decremented instead.

In other words, statement A and statement B are identical in effect as the following respectively:

```
++$a; $b = $a;      # equivalent to statement A
$b = $a; ++$a;      # equivalent to statement B
```

The exponentiation operator calculates the n^{th} power of a number. For example, 4^3, i.e. `4*4*4` is expressed by `4**3`, and the result is 64. Both operands can be floating point numbers.

All the operands discussed in this section take on scalar values only. In other words, they create a scalar context for the operands.

4.2.2 String Manipulation Operators

String manipulation operators include the following:

Operator	Description
x	String repetition operator
.	String concatenation operator

The string concatenation operator is used to concatenate two strings. In other words, it glues two pieces of string together. For example,

```
"hello " . "guy"
```

results in the string "hello guy".

You can concatenate as many pieces of string as you wish by using a series of concatenation operators together, like this:

```
$username . ", your disk quota is " . $quota . " Megabytes."
```

In general, the concatenation operator dictates that both operands must be strings, and numeric operands would be converted to string form before concatenation (still remember that Perl does type conversions internally if necessary?). However, Perl can become highly confused about whether you are using the concatenation operator or the decimal point if both operands are numeric literals. For example:

A. `print "1"."1";` (return: "11")
B. `print "1".1;` (return: "11")
C. `print 1."1";` (return: "11")
D. `print 1.1;` (return: 1.1)
E. `print 1 .  1;` (return: "11")
F. `print 1.  1;` (return: error!)
G. `print 1 .1;` (return: "11")

In case A, B and C, Perl thinks that the dot represents the concatenation operator because one or more operands is a string literal. Note the difference between cases D and E. Although whitespace does not influence how Perl interprets an expression in general, this is not the case as shown in this example. In case D, because the dot follows the first 1 immediately, Perl thinks that you would like to use the decimal point, and returns the numeric value 1.1. In case E, however, because there is a space before the dot, Perl thinks that you would like to use the string concatenation operator, and glues them up for you. So is case G.

However, in case F, Perl thinks that you would like to supply a floating-point number like in case D, but you supply it with two numbers with no comma in between (note that 1. is identical to 1),

so this is a syntax error! You get the same error if you replace case F with `print 2 3;`. However, because it is nonsense (although allowed) to concatenate two literals, just take this as some extra information and little attention can be paid to it.

In scalar context, the string repetition operator returns the string specified by the left operand repeated the number of times specified by the right operand. For example,

```
$str = "ha" x 5;
```

results in the string "hahahahaha" being assigned to `$str`.

In list context, if the left operand is a list in parentheses, it repeats the list the specified number of times. Examples:

```
@array = ("a") x 3;     # or even (a) x 3 will work, but not a x 3
```

results in the list `("a", "a", "a")` being assigned to `@array`.

```
@array = 3 x @array;
```

replaces all the elements with the value 3. Note that the second operand always has a scalar numeric context.

4.2.3 Comparison Operators

In Perl, there are two sets of comparison operators. The first set compares the operands numerically:

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<=>	general comparison

The second set compares the operands stringwise:

Operator	Description
lt	less than
gt	greater than
le	less than or equal to
ge	greater than or equal to
cmp	general comparison

At this point it is important to tell you one behaviour of Perl. As you know, Perl differentiates only scalar and list data. That implies that Perl is quite ignorant about whether the value of a given variable (or a scalar element of a list) is a string or a number. This is why we have two sets of comparison operators defined in Perl, in this way the programmer should choose the appropriate

set of comparison operator for comparison.

Because of the fact that Perl does not differentiate string and numbers much, it allows you to use a string wherever a number is required, and vice versa. However, this involves a conversion that I have to explain it here. This behaviour is not specific to comparison operators only, but a general rule that you have to bear in mind at all times when you are writing your own scripts. Consider the two statements below:

```
A.   print "23.1abc" + 4;
B.   print "23.1abc" .  4;
```

The addition operator (+) requires a numeric context (i.e. it requires numeric values as operands). Therefore, Perl extracts the leading numeric portion of the double-quoted string until a non-numeric character is encountered, and converts this portion into a number (which yields 23.1 in this example). 4 is then added to this number, thus yielding 27.1 for the first statement. But what if the string started with non-numeric characters? Simply a 0 is returned. Also, if the string-to-number conversion involves a string that contains any non-numeric characters, Perl will display a warning message if you have warnings turned on.

The concatenation operator (.) requires a string context. Therefore, Perl converts the second operand (4) into a string and is then appended to the end of the first operand ("23.1abc"), thus yielding the output "23.1abc4". This is how Perl automatically converts between numbers and strings to and forth as needed.

Comparing two numbers is easy, but what about two strings? How does the computer compare two strings, possibly with non-alphanumeric characters in it? Perl compares two strings by comparing the **ASCII Code** of each individual character in the string. As internally the computer only recognizes numbers, a way of representing characters with numbers have to be devised. ASCII representation is one of the several schemes available, and is well adopted. You can go to this website to consult the ASCII Table, which contains the characters with their associated ASCII code. ASCII codes are in the range 0 - 127, and there is an extended set in the range 128 - 255 which is not well supported on many systems. Don't ask me why the characters are assigned in this way. The ASCII table was defined as such, and there's little significance about its origin anyway. Let's compare the two pieces of string in this example:

"Urgent", "agent"

Perl compares character by character. First compare the first character of the two strings, 'U' and 'a'. 'U' has the ASCII code of 85, while 'a' is 97. Because 97 is greater than 85, Perl decides that the latter character is greater, and thus "agent" is greater than "Urgent". In the case of comparing "tooth" and "toothpicks", the longer one prevails for obvious reasons.

After you have learned how Perl handles string and numeric comparison in general, it's time to look at how you specify the comparison with Perl. That's easy, because you merely put the appropriate operator in between the two operands. For example, to test if "agent" is less than "Urgent" stringwise, simply specify `"Urgent" lt "agent"`, and the result is true. Note that Perl does not have the intrinsic Boolean type, so if you pass the result directly to the `print` function, it is customary for Perl to output 1 for true, and "" (an empty string) for false. The best way to have the test result displayed properly is by using the conditional operator `?:`. I will talk about this shortly later on in this chapter, but this is how you can do it:

```
print "Urgent" lt "agent" ? "true" : "false";
```

causes the word "true" to be printed if the test is true, and "false" is printed if otherwise.

Here is another example showing how numeric comparison may be used to decide on which block of code to be executed:

```
if ($score >= 90) {
    print "Well done. Your score is $score.\n";     # A
} else {
    print "Work hard. Your score is $score.\n";     # B
}
```

In the next chapter you will learn how to use the conditional statement like `if` as presented in this example. Basically, the concept is simple. If the value of `$score` is greater than or equals 90, statement A is printed; statement B is printed otherwise.

Be aware that you may get different results if you compare two pieces of scalar data numerically or stringwise!

The <=> and `cmp` operators can be regarded as general comparison operators. Because of the shape, they are sometimes referred to in Perl manpages as *spaceship operators*. <=> compares numerically while `cmp` compares stringwise. The action of these two operators are similar, and I shall take <=> as an illustration.

The characteristics of <=> is as follows. Denote the left operand as `$a` and the right operand `$b`. If `$a < $b`, the result is -1. If `$a > $b`, the result is 1. If both operands are equal, that is, `$a == $b` (see below), the result is 0. This is a handy operator to quickly establish a trichotomy by determining in a single operation whether a number is greater than, equal to or less than another. Despite its power, it is seldom used in practice. It is mostly used with the `sort( )` function to sort a list of scalars in a convenient manner.

4.2.4 Equality Operators

Equality operators include the following:

Operator	Description
==	equal (numeric comparison)
!=	not equal (numeric comparison)
eq	equal (stringwise comparison)
ne	not equal (stringwise comparison)

Similar to the case for comparison operators, we have two sets of equality operators. One set for numeric comparison, the other set for strings. Equality operators can usually be regarded as part of the comparison operators, but some books may prefer to classify them into two categories. There's actually little point to argue which approach is better, as different book authors take different views. Equality operators compares two pieces of scalar data and return a Boolean value (again, scalar value instead in Perl) that indicates if the two pieces of data are identical.

The first two operators compare numerically, while the remaining two compare stringwise. For the equal operators (==, eq) they return true if the two operands are identical, false if otherwise. The inequality operators (!=, ne) have an opposite sense, they return false if the two operands are identical, true if otherwise.

The equality and comparison operators we have covered so far are concluded by these four examples:

```
A.  'true' == 'false' # true !!
B.  'add' gt 'Add' # true
C.  'adder' gt 'add' # true
D.  '10' lt '9' # true
```

In example A, == requires a numeric context, thus both strings are converted into 0. Both sides are equal and evaluates to true (although you would receive a warning if **-w** is enabled). In example B, Perl will stop after checking the first character since 'a' is greater than 'A'. Beware that in the ASCII table capital characters have smaller ASCII codes than the small letter counterparts! In example C, because the first three characters are the same and Perl cannot yet deduce whether 'adder' is greater than 'add' the longer string shall be considered greater. In example D, since we compare with lt, '1' is less than '9', therefore, the comparison evaluates to true. This example and example A illustrate why in Perl we need 2 sets of comparison operators. Because Perl is loosely typed and does type conversions automatically, there should be a method for Perl to know whether you would like to compare them as numbers or strings.

4.2.5 Logical Operators

Logical operators include the following:

Operator	Description
|| or	Logical OR
&& and	Logical AND
! not	Logical NOT, i.e. negation
xor	Logical XOR — Exclusive OR

The logical operators performs Boolean logic arithmetic. We have seen how to do a test using the comparison and equality operators. But what if you would like to carry out two or more tests and check if all of them are true? Boolean algebra can do it rather easily. However, it is out of the scope of this tutorial somehow for me to teach you the specifics of Boolean algebra, and I would focus on how to use the Perl logical operators only.

You may discover that there are two sets of OR, AND and NOT operators. ||, && and ! refer to the C-Style version. If you know C/C++, these operators would look familiar to you, and are continued to be supported in Perl. Perl also has its own set, consisting of or, and, not and xor. Note that Perl gives you an extra xor logical operator, that is not available in C/C++. The two sets differ only by precedence (which you will learn in the next section). The C-style operators have higher precedence, while the Perl operators have the lowest precedence among all the Perl operators.

In the above example, we would like to see if the results of both tests are true. We can then use either logical AND operator to do it. The result would be true only if both tests are true, and

false if otherwise. The logical OR operators return false only when both tests are false, and true if otherwise. The logical NOT operator toggles the truth value. For example, `!(13 < 25)` is false, because it inverts the truth value of `13 < 25`, which is true.

The exclusive or operator returns true if exactly one of the two operands is false. That is, one is true while the other is false.

Here is the truth table of the logical operators we have covered so far:

test1	test2	and &&	or ||	xor
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

test	not !
true	false
false	true

Table 4.1: Truth table of various Perl logical operators

Here are several examples to conclude:

`(4<8) && (16<32)` (return: true)
`(4<8) xor (16<32)` (return: false)
`(4<8) || (16<10)` (return: true)

Another important behaviour I haven't told you yet concerning logical operators is the short-circuiting property. Take `and` as an example. Given the expression

```
(4 > 6) and (5 < 7)
```

you can tell by merely looking at the first expression that the whole expression is false, regardless of the value of the second expression. Similar to the case for the `or` operator, if the first operand evaluates to true already, it is not necessary for Perl to examine the second expression. Therefore, Perl will just ignore that expression and do NOT even attempt to evaluate it. This behaviour is known as short- circuiting. The logical AND as well as logical OR operators support short-circuiting. `xor`, for example, cannot short-circuit because its nature requires the value of both expressions be examined.

Short-circuiting is significant because it eliminates several runtime errors, in particular in the following example we will not get the "division by zero" error because of short-circuiting:

EXAMPLE 4.1 Safe Division

```
1 #!/usr/bin/perl -w
2
3 # safediv.pl
4 # Safe division which detects division-by-zero
```

```
5
6  print "== Safe Division ==\n";
7  print "Please enter the dividend > ";
8  chomp($x = <STDIN>);
9  print "Please enter the divisor > ";
10 chomp($y = <STDIN>);
11
12 $quotient = ($y || undef) && $x/$y;
13 if (!defined $quotient) {
14     print "Division by zero!\n";
15 } else {
16     print "$x / $y = $quotient\n";
17 }
```

The core of the program is on line 12, which appears to be a little bit complicated. This example makes use of short-circuiting property of && and ||. Let's consider two cases, when $y is non-zero and zero.

Consider the case if $y is non-zero. First $y is evaluated. Because it is not zero, the short-circuiting property of || causes the second operand `undef` not to be evaluated. Therefore, the first operand of && is $y itself. Because it is non-zero, the second operand has to be evaluated, which calculates the quotient, which is assigned to `$quotient`.

If $y is zero, then the second operand of || has to be evaluated, which is `undef`. The operand of && is thus `undef`. Because this value is interpreted as a false value in Boolean context, the second operand of && is not evaluated and thus `undef` is assigned to `$quotient`.

Subsequently, a check of whether `$quotient` is `undef` is sufficient to tell whether "division by zero" occurs. Depending on this result, either an error message or the quotient will be `printed`.

4.2.6 Bitwise Operators

Bitwise operators refer to the following:

Operator	Description
<<	Binary shift left
>>	Binary shift right
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT

The first two operators are the binary shift operators. The two operands of these operators must be integral. As you may know, numbers are represented in binary form internally. The left operand is the number to be operated on, while the right operand is the number of bits to be shifted. Let me explain this with the help of an example.

Say you would like to perform `60 >> 2`. First we convert 60 into binary notation, that is 111100_2 (subscript 2 means representation in base 2, that is, binary). In this example we intend to shift 2 bits

to the right, that means the two least significant bits (the bits on the far right) are removed, and thus resulted in 1111_2, which is 15_{10} in decimal notation, so 15 would be returned.

We have used the binary shift right in the example. On the other hand, the binary shift left operator does the opposite. It shifts the integer specified left a specific number of bytes, filling the least significant bits with 0. Observant readers may notice that each binary bit shift to the left actually multiplies the number by 2. Try it.

The remaining four operators compare the two integral operands bit by bit. As an example, to extract the 4 least significant bits of the number `$num`, we can use the expression `$num & 15`. The returned value is the integral expression of the last 4 bits of `$num`. The reason is quite simple. The binary representation of 15 is 1111_2. Say `$num` takes the value of 37 (100101_2). The operation is shown below:

```
   1 0 0 1 0 1
&  0 0 1 1 1 1
---------------
   0 0 0 1 0 1
```

The bitwise AND operator compares each bit. If both corresponding bits are 1, the resulting bit is 1; otherwise 0. The above example demonstrates a technique known as **bit-masking**, and 15 is the mask in this example. To extract specific bits, just set up a mask with the corresponding bits setting to 1, and 0 for other bits that we are not interested in. Note that although 15 is only 4-bit long, you may consider that it is extended automatically to 6 bits, with the two most significant bits set to 0.

The other bitwise operators have the same semantics as their logical operator counterparts except the truth value is represented by 1 and 0 for true and false respectively. Therefore, the details of which are not repeated here.

Bitwise operators are in general rarely used in most scripts. They are usually only used in applications such as cryptography or binary file access.

4.2.7 Assignment Operators

Assignment operators refer to the following operators:

Operator	Description
=	Assignment operator
+= -= *= /= %= **=	Arithmetic manipulation with assignment
.= x=	String manipulation with assignment
&&= ||=	Logical manipulation with assignment
&= |= ^= <<= >>=	Bitwise manipulation with assignment

I have mentioned quite a lot about the ordinary assignment operator = in the previous chapters already. Now consider this example:

```
$num = $num + 15;
```

You should understand what this means, do you? The value stored in `$num` is added to 15, and the result is again assigned to `$num`. The net effect is to increment the value of the variable by 15. However, some people may prefer that manipulation and assignment be accomplished by one, instead of two operators. Therefore, Perl recognizes several shorthand notations as shown above.

In general, if you can write a statement in this format:

```
op1 = op1 operator op2;
```

you can have a shorthand version like this:

```
op1 operator= op2;
```

For instance, the example above is identical to `$num += 15;` However, note that not all the operators mentioned above have such a shorthand version. Look at the list above for all the recognized shorthand operators. As an example, to invert all the bits in an integral scalar variable (that is, on a technical parlance, to assign the "1s complement" to the scalar variable), we have to write

```
$num = ~$num;
```

4.2.8 Other Operators

Here I shall introduce to you some other operators that do not fit in any of the above main categories. The operators that would be covered here include the **conditional operator** and the **range operator**. The comma operator and => have already been described in detail in the previous chapter. Putting aside the builtin functions that may be considered operators in Perl, the remaining operators are `->`, `=~` and `!~`. I have chosen to defer mentioning the remaining operators because they are related to some later topics and I would cover them in those sections. The first one, the arrow operator is similar in some sense to the pointer-to-member operator in C. This would be introduced in the references chapter. The other two are used for pattern matching with regular expressions.

The conditional operator `?:` is a ternary operator. In other words, it has three operands. The syntax is as shown below:

```
test-expr ? expr1 : expr2
```

The first operand (`test-expr`) is an expression. If this expression evaluates to true, `expr1` will be returned by the conditional operator; otherwise, `expr2` is returned. You are likely to use conditional operators quite often in the future because it is considered too "bulky" to use the `if-else` structure all the time. This is an example using `if-else` to compare the values of `$a` and `$b`, and assign the smaller value to `$c`:

```
if ($a < $b) {
    $c = $a;
} else {
    $c = $b;
}
```

By using the conditional operator, this 5-line structure (although no one will stop you from writing all this on one line) can be transformed into the simpler statement:

```
$c = $a < $b ? $a : $b;
```

The following example is extracted from the perlop manpage which demonstrates how the list context around the conditional operator propagates to `expr1` and `expr2`.

```
$a = $ok ? @b : @c;
```

In this example, depending on the value of `$ok`, the number of elements of either `@b` or `@c` is returned and assigned to `$a`. First, the assignment to `$a` creates a scalar context around the conditional operator. Therefore, the conditional operator is expected to return a scalar value, and so the two possible values to be returned will be evaluated in scalar context.

Although the conditional operator is inherited from C, there is one important behaviour that is unique to Perl. The conditional operator can be assigned to if both `expr1` and `expr2` are legal lvalues. This is an example:

```
($whichvar ? $var1 : $var2) = $new_value;
```

In the next section, "Operator Precedence and Associativity", we would see an example on how you may obtain unexpected results involving the conditional operator because of operator precedence.

The behaviour of the range operator `(..)` depends on the context around the operator. The range operator actually consists of two disparate operators in list context and scalar context. The range operator in list context is more frequently seen, so I will cover it first.

I have introduced the range operator in list context briefly in the previous chapter, when I described how you can construct an array. The range operator in list context returns an array consisting of the values starting from the left value, with the value of each subsequent element incremented by one until the right value is reached. For example, `@array[0..2]` is synonymous with `@array[0,1,2]` to return an array slice. Note that both the left and right values would be converted to integer by chopping off the decimal portions in case they are not already integers.

Not only integers can be used in the range operator, you can experiment with alphabets as well. For example, ('a' .. 'z') and ('A' .. 'Z') are two lists representing small letters and capital letters, respectively.

4.3 Operator Precedence and Associativity

Now we have learned several operators, so it's time for us to put them together. It is not uncommon that a given statement contains more than one operator. Operator precedence and associativity arise as a result. Recall that in elementary Maths class teachers teach us in a mathematical expression involving several arithmetic operators, multiplication and division shall be performed before addition and subtraction. For example, $3+6\times 7$ is interpreted as $3+(6\times 7)$, not $(3+6)\times 7$. In this case, we say that multiplication and division has higher **precedence** than addition and subtraction operators. Because multiplication has a higher precedence than addition, the multiplication

operation, involving the two operands 6 and 7, is performed first, and then the result 42 is added to 3, getting 45 as the result.

In Perl, because there are many operators, the rules of precedence is far more complicated. In order to describe the relative precedence among the operators, most programming languages would use an operator precedence table to show the relative precedence of the operators, and Perl is of no exception. Table 4.2 shows the operator precedence and associativity table, which you can obtain from the perlop manpage.

Associativity	Operators
left	Terms and list operators (leftward)
left	`->`
nonassoc	`++ --`
right	`**`
right	`! ~ \ + -` (unary)
left	`=~ !~`
left	`* / % x`
left	`+ - .`
left	`<< >>`
nonassoc	named unary operators
nonassoc	`< > <= >= lt gt le ge`
nonassoc	`== != <=> eq ne cmp`
left	`&`
left	`| ^`
left	`&&`
left	`||`
nonassoc	`.. ...`
right	`?:`
right	`= += -= *=` etc. (assignment operators)
left	`, =>`
nonassoc	List operators (rightward)
right	`not`
left	`and`
left	`or xor`

Table 4.2: Operator Precedence and Associativity Table

The operators are arranged in order of decreasing precedence. That is, the operators at the top (Terms and List Operators) have the highest precedence, while the operators at the bottom (`or`, `xor`) have the lowest precedence. Operators on the same line have the same precedence.

The first column lists the **associativity** of the operators. Associativity is useful when there are several operators of the same precedence in a statement. In this situation, the order of evaluation of these operators depends on the associativity. If the associativity is right, then the rightmost one would be evaluated first, then the one on the left, and so on. That's why cascaded assignment is possible. The idea is the same for left associative. You may also find that some operators are labelled "nonassoc" (non-associative). The operators are classified as non-associative if the order of evaluation is not important, or not applicable for other reasons. In general, you don't have to worry about the order of evaluation of the non-associative operators really much.

To demonstrate the effect of operator precedence and associativity, let's go over several scripts here.

```
#!/usr/bin/perl -w

$, = "\n";
$a = 13, $b = 25;
$a += $b *= $c = 35 * 2;
print "\$a == $a, \$b == $b, \$c == $c";
```

This one is practically easy. I would use it to illustrate how precedence and associativity works. The problematic statement is on line 5, and you shouldn't have problems with the rest of the script, so I am focusing on this line only. To deal with this statement, first try to locate the operators concerned. The operators, by scanning from left to right, are:

```
+=  *=  =  *
```

Now locate the operators with the highest precedence, and it is * in this case. Therefore, multiplication would be performed first. The operands of this operator are 35 and 2, so this multiplication yields 70. The remaining are all assignment operators having the same precedence, so we look at their associativity. Because the associativity is right, the rightmost one is evaluated first, which is =. The operands are $c and 70 (the value of the expression 35×2), so 70 is assigned to $c. Then, `$b *= 70` is evaluated, and $b is eventually assigned 1750 (70×25). At last, `$a += 1750` causing $a to be assigned 1763.

There is not much confusion here, but let's consider another more tricky example, of which the result is not necessarily apparent at first glance.

```
#!/usr/bin/perl -w

$, = ", ";
$a = 1, $b = 0;
print $a >= $b ? $b : $a += 6, $a;
```

I first saw a similar example from a C++ book, and I adapted it to become a Perl script. This is a really notorious example, because it includes `?:`, which easily leads to unexpected result if you are not careful enough. I hope you could understand my explanation as this example, though seemingly short, is so notorious that it is rather difficult for me to explain it well.

Now let's have a quiz: without actually running it with your perl interpreter, try to deduce the printout of this example. Let's give you two choices:

A. 0, 1
B. 6, 1

(don't look at the answer below before you have an answer in your mind)

If you choose option A, I'm sorry, you're wrong. But don't despair, as many people share the same mistake as you do, and it is common and understandable for people to make mistakes. Although

novices are more prone to make mistakes, nothing will stop veterans from making mistakes as well. Most computer programs we are using have bugs. Some bugs are obvious, but there are many more hidden ones which are not normally revealed unless your program gives unexpected output. Trust me, debugging is going to occupy most of your development time, and is the most tedious job for all programmers. If you choose option B, congratulations, you are correct. Hope that you did not get the correct answer by sheer guesswork. Anyway, let's look at how we arrive at the answer.

If you choose option A, you may have the impression that += has a high precedence, but this is not the case. In this example, `?:` also exists, which has a higher precedence than +=, so `?:` is evaluated before +=. Here comes the trouble. The operators, scanning from left to right, are:

```
print  >=  ?:  +=  ,
```

`print` is a list operator, and has the highest precedence. So Perl looks on the right hand side for its parameters. Then Perl sees >=, which has the second-highest precedence. So `$a >= $b` is then evaluated, yielding true. Among the remaining operators, `?:` is the highest. The problem is, what are its operands? As you have learned, this operator has three operands, the so-called "test" part (that is `$a >= $b`, and have found to be true), the "true" part (`$b`), and the "false" part. But wait! What is the false part? `$a` or `$a += 6`? Its the former one, in the contrary to what you may have expected.

As Perl scans from left to right to find the false part, it encounters the += operator, which has a lower precedence than `?:`, and Perl stops and claims that the false part is `$a`. Then Perl knows what to do, so evaluates the conditional operator, returning the true part, that is `$b` (that holds the value 0). Anything left? Yes, we still have the dangling `+= 6` part! It is tricky that the conditional operator returns an lvalue if it can, and so `$b += 6` is evaluated, causing `$b` to hold the value 6 instead. We have finished evaluating the first argument to the print operator, and the second operand is `$a`, which has been untouched. That's how we arrive at the answer "6, 1".

Troublesome enough? Two long paragraphs just to explain one statement. If you are still scratching your head, try to read these two paragraphs repeatedly until you understand. So you see, operator precedence and associativity can trip you up if you are not careful.

But what if this is not your intention? What if you'd like to have `$a += 6` as your false part? Recall how you do this in arithmetic — adding a pair of parentheses. This applies to Perl as well. This is easily done as this:

```
print $a >= $b ? $b : ($a += 6) , $a;
```

In Perl, parentheses are treated as "terms", which has the highest precedence in the precedence table. This ensures that all expressions in parentheses are evaluated before other operators do. Parentheses is the cure for those who enjoy writing complicated statements but wouldn't like to memorize (or consult) the operator precedence table. Adding parentheses appropriately not only eliminates a great deal of effort when you or other programmers read your code at a later time, many unexpected errors could be eliminated as a result. But sometimes parentheses do not add much clarity to your code. Here's an example:

```
print(lc(shift(@MyArray)));
```

In this example, all but `print` are named unary operators. Because named unary operators have only one parameter, in general there is not much confusion if you just omit the parentheses. In conclusion, use parentheses wherever appropriate. Feel free to insert parentheses as you see fit to minimize confusion but don't overdo it.

There are several points to note arising from this example. First, although looking naïve to mention here, whitespace is NOT a determinant of the order of evaluation. You definitely cannot write the above statement in this way and expect it to work the same way the parenthesized version does:

```
print $a >= $b ? $b :$a+=6   , $a;
```

This is actually the same as line 5 in the example! So it returns "6, 1", not "0, 1". This is because when the Perl interpreter loads the script, it parses it and removes all intervening whitespace automatically.

Second, you have to be careful about the use of parentheses. Consider these examples:

```
A.  print 1+2+3;
B.  print (1+2+3);
C.  print 1+(2+3);
D.  print (1+2)+3;
E.  print(1+2)+3;
```

In the above examples, statements A–C would give the correct answer, 7, while the last two would give 3. Statements A and B have nothing special. In statement D, however, because the opening brace is placed immediately after the operator, Perl thinks that this pair of parentheses contain all the parameters to be sent to the operator. In this case, `1+2` is its only argument, so 3 is printed. Because the `print` operator returns nothing, thus leaving the dangling part "+3" for Perl to evaluate. Since this is a useless addition operation, a Perl warning would be displayed if you have warnings enabled. Statement E is the same as D (recall that whitespace does not matter?).

For statement C, because you do not have an opening brace immediately following it, Perl does not think that there are parentheses to contain the parameters as in statement D. The parentheses in statement C is treated in the normal way, that is, evaluated first, and then the lower-precedence operators.

Third, notice that in the operator precedence charts there are two entries for list operators. List operators (leftward) in this tutorial (and in the perlop manpage) refer to the name of the list operator concerned. Such a high precedence allows Perl to know where such operators occur, so that it knows where to look for their parameters. On the other hand, list operators (rightward) applies to the comma-separated expressions to pass to the list operator concerned as parameters. The extremely low precedence of this part implies that you do not normally need to put parentheses around the parameters. This part acts figuratively like a basket containing all the parameters. The only operators having lower precedence are the operators `not`, `and`, `or` and `xor`. For example,

```
open HANDLE, "$path/.bash_history" or die "Can't open file\n";
```

Because `or` has a lower precedence than list operator (rightward), the `or` part is not treated as part of the operand for the `open( )` operator. In the example, if the `open` operation fails, it returns `undef`,

which is interpreted as false. Because short-circuit evaluation is not possible, the second part is executed, which outputs the error message and terminates the script. You would learn how to open files later on in this tutorial.

4.4 Constructing Your Own `sort()` Routine

In the last chapter you have had an overview of using the `sort()` function to sort a list of values. I also provided you with a list of common search routines. By now you should have sufficient knowledge to understand the rest of the story.

The principle is pretty simple but rather difficult to visualize. You override the default sort criteria by defining the ordering criteria in the code block of the `sort()` function. Two special variables `$a` and `$b` are defined in the block. In order to sort a list of values, the sort routine needs to establish the total ordering of any two arbitrary items in the list. To do so, it assigns the two items to `$a` and `$b` arbitrarily. The block should perform some comparison operations in the block based on these two values. If evaluation of the block yields a scalar value that is negative, the value held by `$a` comes before that of `$b`. If the result is positive, the value held by `$b` comes before that of `$a`. If the result is zero, then the two values should be considered equal. The <=> and `cmp` operators are usually used. The following is an example which illustrates numeric descending sorting.

Assume the unsorted list is `(3, 1, 4)`. To sort in descending numerical order, an appropriate expression which satisfies the above characteristics is

```
$b <=> $a
```

The following table illustrates how you can try to verify if the comparison works. $3 \rightarrow 1$ below means 3 comes earlier than 1 in the resulting list.

$a	$b	$b <=> $a	Ordering
3	1	-1	$3 \rightarrow 1$
1	4	1	$4 \rightarrow 1$
3	4	1	$4 \rightarrow 3$

Table 4.3: An Example Illustrating `sort()`

Therefore, the ordering $4 \rightarrow 3 \rightarrow 1$ is established.

Summary

- Operators carry out certain operations on data, known as operands.
- An expression consists of a combination of operators and operands.
- Unary, binary and ternary operators take on one, two and three operands respectively. List operators can take on a variable number of operands.
- In Perl, functions are simply treated as unary or list operators.
- Arithmetic operators carry out arithmetic calculations.

- Assignment operators are used to assign scalar or list value to a data structure.
- Comparison operators compare two pieces of scalar data and returns a Boolean value.
- Equality operators are specializations of comparison operators which test if two pieces are considered equal.
- Bitwise operators can be used to perform binary arithmetic.
- Logical operators operate on Boolean values.
- String manipulation operators involve string concatenation operations.
- Operator precedence and associativity determines the order in which operators are evaluated in an expression.
- The `sort()` function sorts a list of scalar values. Ordering is determined by the return value of the code block provided in the `sort()` function.

Chapter 5

Conditionals, Loops & Subroutines

5.1 Breaking Up Your Code

So long you have been writing your programs in one piece. You are totally allowed to carry on with this practice, however, lumping everything in one piece is often considered undesirable for the following reasons:

Ease of maintain A single source file with tens of thousands of lines is for sure not easy to maintain. First, navigating around in the source file is messy — page up and page down keys are unlikely to be effective for files of this size and to locate a certain section of code much traversal is necessary.

Variable Conflicts Variable management is not a trivial affair in practice when the program is a large one. It is easy for us to keep track of the variables in use for short programs we have been writing for now, but shortly when the code base grows in size, it becomes increasingly likely that variable name clash occurs at certain parts of the source code. That is, you use a variable that is defined somewhere else but you are unaware of it. This may give rise to some unexpected behaviour that are difficult to debug. Object-oriented programming is a desirable solution. We would discuss the concepts of scope and packages, which are fundamental to object-oriented programming that would be covered in the next chapter.

5.1.1 Sourcing External Files with `require()`

The `require()` function can be used to source external files. It acts in this way — when execution reaches a `require()` statement, it first checks if the file has already been `required`. This is to avoid multiple inclusions and potential file inclusion loops (for example, A includes B, and B includes A etc.).

The predefined variable `@INC` stores path prefixes the Perl file inclusion system use to find the file to be sourced. You may type the following command at the command prompt to output the content of `@INC`:

```
perl -e 'print map {"$_\n"} @INC;'
```

On my Linux system, the list of paths is, in ascending order,

```
cbkihong@cbkihong:~/public_html> perl -e 'print map {"$_\n"} @INC;'
/usr/lib/perl5/5.8.0/i686-linux-thread-multi
/usr/lib/perl5/5.8.0
/usr/lib/perl5/site_perl/5.8.0/i686-linux-thread-multi
/usr/lib/perl5/site_perl/5.8.0
/usr/lib/perl5/site_perl/5.6.1
/usr/lib/perl5/site_perl
.
```

There are 7 entries. The last entry is a single dot denoting the current directory. Note that the paths listed are likely to be different on each system. They are fixed in the Perl installation process.

`require()` is an overloaded function serving three purposes:

- ⋆ If the argument is a number, it checks if the perl interpreter version (`$]`) is greater than the version specified as the argument. This use generally appears in Perl modules to specify the earliest perl interpreter version with which the module can be executed;
- ⋆ If it is a bareword, that is not enclosed in quotation marks, it is assumed to be the package name of a module with extension ".pm" (see Chapter 7) and any occurrences of '::' are converted to '/' on Unix-variant OSes or '\' on Windows or MS-DOS. The result is a relative path to a Perl module;
- ⋆ If the argument is a quoted string, it is treated as the relative path to the file to be included.

For the last two cases, the relative path resulted is appended to each of the paths in `@INC` in turn to form a complete path and `require()` checks if a file exists at this location. The first file that is found is used. As you can see, the system modules directories are searched first, and at last the current directory is searched.

If the search yields an existing file, the relative path is added to `%INC` as the key while the complete path formed as the value. The `%INC` hash variable is maintained by Perl which contains a list of files included. Whenever you invoke the `require()` function, it checks this hash to determine if the file specified already exists in `%INC`. If it does not, an entry is added to it; otherwise, the inclusion is silently bypassed.

For example, say you have a source file which has the following `require()` statements:

```
require 'myfile.pl';
require MyModule;
```

The `require()` function will append the key-value pairs

```
(
    'myfile.pl' => '/home/cbkihong/somedir/myfile.pl',
    'MyModule.pm' => '/home/cbkihong/somedir/MyModule.pm',
)
```

to `%INC`. Consider the situation if inside the module `MyModule` it also tries to `require 'myfile.pl'`, because an entry with the key 'myfile.pl' already exists in the hash, this inclusion is bypassed. Therefore, multiple file inclusions can be detected and avoided.

If the file `required` does not return a true value (as a recapitulation — anything other than 0, an empty string or `undef`), `require()` fails with an error. Perl module authors may utilize this characteristic to abort the scripts using the module in case of errors. Therefore, you ought to see in later chapters that external files usually terminate with the statement `1;` which returns the true value required.

The description above is a rewrite of the `require()` manpage. It is easy to understand and you are advised to read it for further information. In Chapter 7 you would learn the `use()` operator which is a wrapper of `require()` and is used to load Perl modules.

5.2 Scope and Code Blocks

5.2.1 Introduction to Associations

Consider this simple assignment statement:

```
$fruit = "apple";
```

How would you describe this statement? The string "apple" is assigned to the scalar variable named `$fruit`. Fine, this is how the statement is interpreted literally, but this description is not sufficient as you progress through the following chapters. A better and more in-depth way of expressing this statement is that a data object with the scalar value "apple" is created, and the name "fruit" is now associated with this data object. Why do we have to bother with this expression? Because up to now we have been writing programs with each data object associated with one and only one name, which is the name of the variable. However, in this chapter we would see that due to scoping rules the same name at different parts of a program can be associated with different data objects. This **association** is also known as a **binding**. In the next chapter we would introduce to you the possibility of establishing additional associations to a data object, which are references. Therefore, from now on you need to have a clear separation of the data object and its name associations.

5.2.2 Code Blocks

The idea of confining the scope of associations arose from the fact that not all variables need to be valid throughout the lifetime of the program. This is particularly important in subroutines, which are self-contained reusable units containing a sequence of statements.

The **scope** of an association refers to the region in a program in which the association is visible. Up to the present, we have been writing very simple scripts with scalar variables, arrays and hashes which, once created, could be used anywhere in the script. What we have been using in our examples so far are all global variables, in the sense that all associations are established during the compilation phase before execution, and are not destroyed until the program terminates because these variables are resolved from the symbol table, which exists as long as the program is in execution. Note that once you have created an association there is no way for you to manually destroy it. An association will only be destroyed when the scope in which it resides terminates. In this sense, a global variable has a global scope. Excessive use of global variables is considered a poor programming practice, because as your program becomes increasingly complicated, it is not impossible that your program may involve dozens to even hundreds of variables. Perhaps you may need some of them to be accessible throughout the whole script, but this is not generally the case.

In particular, many variables are simply used to store data temporarily, for example, as a counter in a loop (as you will see later in this chapter). Introduction of limited scopes is seen as a cure to the problem.

Scopes are defined in terms of **environments**. An environment is created by means of a **code block**, or simply, a block. A code block simply consists of a sequence of statements that makes up an environment, and is delimited by curly brackets (but the mere presence of { } doesn't necessarily imply a code block — you will see `do{}` and `eval{}` later in this chapter that look like blocks, but they are not strictly one). Code blocks can be nested. In other words, one code block can appear in another code block. Like this:

```
# Environment A
{
    # Environment B
    {
        # Environment C
    }
    # Environment B resumes
    {
        # Environment D
    }
    # Environment B resumes
}
# Environment A resumes
```

As you can see, after { a new environment is started. When } is reached, the current environment terminates, and the parent environment, that is, the environment that was previously in effect is reinstated. For example, when environment C terminates, environment B will resume. With `my` and `local` modifiers that we are going to explore in the next section, you can confine the scope of an association to within the extent of an environment — when the containing code block terminates, the association would be destroyed.

NOTES

Please interpret the above statement carefully. Although all associations are destroyed when the current environment terminates, this does not necessarily mean the objects themselves are destroyed. This is related to the garbage collection mechanism in Perl, which will be covered in the next chapter when we come to references. In particular, if additional references to the object exist, the object will not be destroyed.

5.3 Subroutines

Breaking up your source code into multiple files is not the only way you can make your code more manageable. A complex program can be broken into smaller tasks, each of which carries out a well-defined function. Take an online book catalogue served in a typical library as an example, you can actually split the whole complicated program into smaller parts. For example, these are the

functions that may be implemented in such a system:

- ⋆ Search the catalogue by title, author, keywords, etc.
- ⋆ Allows users to check their circulation record
- ⋆ Allows users to reserve or renew items
- ⋆ Perhaps to send a reminder to a borrower automatically if he forgets to return the borrowed items on time

You should be able to think of many more, but this already exemplifies a program will be easier to write and manage if you break it up into smaller and simpler parts, with each part doing its intended task only. This shows how useful subroutines are. A **subroutine** consists of a sequence of statements defining an environment. Subroutines are just like user-defined functions in contrast to those provided by Perl. There are two types of subroutines — named subroutines and anonymous subroutines. Anonymous subroutines are not given a name while named subroutines are. We shall cover anonymous subroutines in Chapter 6 when we come to the topic of references. From now on, by means of subroutines we refer to named subroutines.

5.3.1 Creating and Using A Subroutine

In general, before we can call a subroutine, we need to declare and define it. Declaring the subroutine makes Perl aware that a subroutine of a particular name exists. Defining means you explicitly describe what the subroutine does by listing the statements to be executed if the subroutine is being called.

In general, subroutine declaration and definition go together. The syntax used to declare and define a subroutine is as follows:

```
sub name [(prototype)] block
```

`block` is the subroutine definition. It is a code block containing the statements to be executed when the subroutine is invoked. The rest is the subroutine declaration. It declares a subroutine with the name `name`. After the subroutine name you may insert an optional `prototype` part which contains a concise specification of the types of parameters to pass to the subroutine.

Here is an example using a subroutine to calculate the sum of a list of scalars (presumably numeric):

EXAMPLE 5.1

```
1  #!/usr/bin/perl -w
2
3  sub sum (@) {
4      # This subroutine takes a list of numbers as input
5      # and returns the sum
6      my $sum = 0;
7      for my $tmp (@_) {
8          $sum += $tmp;
9      }
10     return $sum;
```

```
11 }
12
13 # calculates 0 + 1 + 2 + ... + 100 and prints the value
14 print sum 0 .. 100;   # must be 5050. No doubt.
```

Lines 3-11 contain the declaration as well as the definition of the subroutine `sum()`. Line 3 tells Perl that you are declaring a subroutine named `sum`, while lines 4-10 are the subroutine definition. Subroutines have to be declared and defined before being called in the script. If you place the `print` statement (line 14) before the subroutine declaration, your script simply won't work as expected. This is because the perl interpreter reads your script sequentially. If it encounters the token `sum` before it is declared, Perl will not know `sum` is a user-defined subroutine as Perl cannot find it in the system libraries. That implies you should, in principle, always put subroutine declarations and definitions very early in source files. As you can see in the example, the subroutine is put before the source program that calls it. However, this may not be convenient sometimes. For example, when you have multiple subroutines which are inter-dependent on one another, it may not be convenient for you to find out a proper order of declaring the subroutines. Therefore, Perl allows you to make forward declarations, at which point the subroutines are declared but not defined. Forward declarations are put at the very top to declare the subroutines, and somewhere later on you give the definitions of the subroutines concerned, which can then be placed in any order. In the example, in order to make a forward declaration of the `sum` subroutine, it can be made like this:

```
sub sum(@);
```

This statement tells Perl in advance (before parsing the subroutine definition) that `sum` is the name of a subroutine accepting a variable number of scalars as arguments, and Perl will know to look for the definition of `sum` in the later part of the program. The forward declaration is the same as the declaration line except we replace the block containing the definition with a semicolon.

Although you may not understand the `my` modifier and the `for` loop in the script at the moment, you may ask a question: why do I have to resort to subroutines if I can implement it directly with a `for` loop or a `foreach` loop? Well, note that the subroutine `sum` serves a general purpose. It adds up all the input values and return the result, regardless of the values of the input. Therefore, this subroutine is highly versatile and flexible — it can be used directly without modification on any occasion you would like to evaluate the sum of a list of scalars. If you write the script as follows, it can only be used to sum up all the integers between 0 and 100. Whenever you would like to evaluate a sum of something else, you have to rewrite the code:

```
# calculates 0 + 1 + 2 + ... + 100 and prints the value
my $sum = 0;
for my $tmp (0 .. 100) {
    $sum += $tmp;
}
print $sum;
```

By writing your code in subroutines, you are enforcing **reusability** of your code. Code reusability is important as we wouldn't like to write similar pieces of code again and again. Later on you will learn how to build reusable modules where you can put your subroutines and save them as a file so that whenever you need to use the subroutines in another project, you just need to import the module, and the subroutines can be reused in your new project. This is very convenient. Here is how the `sum` subroutine can be readily applied to evaluate the sum in other situations:

```
print sum values %score;  # print the sum of the values of %score
$avg = sum(@nums)/@nums;  # evaluate the average of the values of @nums
```

Apart from improved reusability, subroutines also help make debugging easier. Once we are certain that a subroutine is correct, we can safely apply it. It is likely that fewer number of errors would be committed when we combine the subroutines to form an entire program than writing it without using any subroutines.

Unlike other languages like C or Java, Perl does not support named parameters. All incoming parameters are combined into one indistinguishable array `@_`. The first parameter is thus `$_[0]`. Therefore, if you have a mixture of arrays and scalars the elements of which would be combined into `@_`. You may use pass-by-reference to be discussed in the next chapter to avoid it.

Also, note that the elements of `@_` are not copied by value. The elements of `@_` are "references" to the data values. They behave like references but do not look like references in terms of syntax. Basically, the idea is when you modify a value of an element in `@_`, the corresponding data object will be modified as well. Consider this example:

```
sub test {
    $_[2] = 5;
    $_[3] = 6;
}

my @a = (1, 2, 3);
my $b = 4;
test(@a, $b);
print join(', ', @a, $b), "\n";     # 1, 2, 5, 6
```

Here, when the third as well as the fourth element of `@_` are updated, the corresponding elements in `@a` and `$b` will be modified. What if we replace `$b` with the literal 4? Now because the data object cannot be modified, it will be an error. In principle, changing the value of parameters silently in a subroutine is a bad programming practice, although sometimes you cannot avoid it. You should document these cases clearly, for example, as comments in the source files.

Because a parameter can be inadvertently modified in a subroutine, in general, you should not use the elements of `@_` directly in scripts. You can use the following technique:

```
sub search {
    my ($myitem, @myarray) = @_;
    # use $myitem and @myarray thereafter
    # ...
}

my @array = (1, 2, 3);
my $searchFor = 2;
search($searchFor, @array);
```

By using the `my` modifier, `$myitem` and `@myarray` are confined to within the `search` subroutine only. Here, the elements are copied by value to `$myitem` and `@myarray`. Therefore, the tie between them and the original data objects no longer exists. Even if you inadvertently modify the values or `$myitem` or `@myarray`, the changes won't be made to the original data objects. Moreover, this

emulates named parameter passing in other programming languages.

A subroutine may return a list of values to the caller. As shown in Example 5.1, this is achieved by the `return` function. A subroutine may return a scalar or a list of scalars. Similar to the case for incoming parameters, multiple arrays or hashes are combined into one single list. Again, you may use references to circumvent this, though.

5.3.2 Prototypes

Recall that while declaring a subroutine you may put an optional prototype specification. It describes the number of as well as the type of parameters in a compact form. The prototype gives Perl a clue as to how the arguments should be handled. Having an accurate grasp of the types of arguments expected is important, as illustrated in a mini case study below.

The prototype specification comprises a sequence of symbols indicating the type of each argument. The symbols should look familiar to you, because they are the same symbols which are used to denote scalar variables, arrays, hashes etc. In front of each symbol you may prepend a backslash \ to indicate the element is to be passed by reference. This is covered in the next chapter.

Symbol	Type
$	Scalar variable
@	Array
%	Hash
&	Anonymous subroutine
*	Typeglob (Reference to Symbol Table entry)

For example, if the prototype is `($$)`, that means the subroutine accepts two scalar variables as parameters. `($$@)` implies the first two parameters to be evaluated in scalar context while the remaining parameters would be grabbed by an array variable. Note that you cannot have something like `(@$)` as the array variable (or hash variable alike) would take up all the input parameters. Always bear in mind that multiple parameters, after evaluating in their respective contexts, are combined together to become one indistinguishable array `@_`.

A "programmer" who claimed to know Perl was asked by his boss to write a subroutine which inserts a list into an array at a certain position. There is already a `splice()` function which can do that for him, so he decided to write a wrapper which calls `splice()` to do the job. The boss, as a user, would like to use the subroutine in this format:

```
insert @array, pos, list
```

which is identical to the syntax of `splice()` except without the length parameter. The "programmer" wrote this:

```
sub insert {
    # !!WARNING!! This does NOT work!
    my (@myarray, $pos, @list) = @_;
    return splice(@myarray, $pos, 0, @list);
}
```

Without even trying it, he handed it to his boss. The boss tried to use it in this way:

```
@array = (1, 2, 3);
insert(@array, 3, 4, 5, 6);
```

It didn't work, and he lost his job. Does that sound too stupid for you? Why doesn't it work? As I have reiterated a number of times already, because all the parameters are combined into a single list when they are passed to the subroutine. You can't really separate them back into the three parameters, because the first argument is an array which due to its "greediness" would take all the elements passed into the subroutine, leaving `$pos` and `@list` undefined. The proper way to do this is:

```
sub insert (\@$@) {
    my ($array, $pos, @list) = @_;
    return splice(@$array, $pos, 0, @list);
}
```

It uses both pass-by-reference with a prototype added to make the types of parameters expected explicit. The use of `@$array` causes the original array to be modified, as we'll cover in the next chapter. As for the prototype, we indicate the parameters are an array reference, a scalar and then a list. If a subroutine has a prototype, Perl will try to evaluate the parameters according to the prototype. Consider the case if the boss uses the subroutine in this way:

```
@array = (1, 2, 3);
insert(@array, @array, 4, 5, 6);
```

This is identical to the previous case. Note that in the prototype the second parameter indicates a scalar is expected. Therefore, a scalar context is put around `@array` which causes the number of elements in `@array` to be passed as the second parameter, which causes the list specified to be appended to the end of the `@array`.

You can also specify optional parameters. Compulsory parameters are separated from optional parameters by adding a semicolon in between. For example, say you have a subroutine whose declaration statement is `sub mysub ($$;$$);`, and you make the following subroutine call:

```
mysub @array, "3", 9;
```

Because `@array` is evaluated in scalar context, `@_` is the list `(scalar(@array), "3", 9)`. The fourth parameter is empty. By using prototypes you can let Perl check parameter types and evaluate the parameters in the correct contexts.

A sidenote about subroutine invocation. Traditionally, subroutines had to be prefixed with the `&` symbol when invoked, and the parentheses are compulsory in this case. For example, `&mysub(@array, "3", 9);` When a subroutine is invoked in this way, the prototype is ignored. Therefore, I recommend not to use the `&` form in general. A few situations where you need to use the `&` form will be covered in the next chapter.

5.3.3 Recursion

Recursion is more of a technique rather than a feature of a programming language. It refers to the practice of tackling a problem through dividing it into smaller sub-problems and tackling them independently. Each of these sub-problems may also be subdivided if necessary. In programming languages, recursion is typically achieved through nested invocation of a subroutine, directly or indirectly. We'll examine recursion with the help of an example.

A palindrome is a sequence of characters that is identical regardless you read it in a forward or backward direction. For example, "dad" and "sees" are examples of palindromes. Here, we tackle the problem of determining whether a given string is a palindrome. For simplicity, we only consider strict palindromes that are symmetric character by character. Phrases like "Madam, I'm Adam" are generally considered palindromes, but we don't classify them as such.

There are (at least) two ways to tackle this problem, namely the iterative and recursive approach. First, we present the source program for the iterative approach:

EXAMPLE 5.2 Palindrome

```
1  #!/usr/bin/perl -w
2
3  # Determining whether a given string is a palindrome.
4  # (Iterative approach)
5
6  sub isPalindrome($);
7
8  print "Enter a string > ";
9  chomp(my $str = <STDIN>);
10 if ($str ne '') {
11     print "$str is ", isPalindrome($str)?"":"not ", "a palindrome.\n";
12 } else {
13     print "The string should not be empty!\n";
14 }
15
16 sub isPalindrome($) {
17     my $string = $_[0];
18
19     my $ctr_l = 0;
20     my $ctr_r = length($string)-1;
21
22     while ($ctr_l <= $ctr_r) {
23         # To do case-insensitive comparison, convert both to lowercase if ¶
               applicable
24         my $leftchar = lc substr($string, $ctr_l, 1);
25         my $rightchar = lc substr($string, $ctr_r, 1);
26         if ($leftchar ne $rightchar) {
27             return 0;
28         } else {
29             $ctr_l++;
30             $ctr_r--;
31         }
32     }
```

```
33      return 1;
34  }
```

Then the recursive approach:

```
1   #!/usr/bin/perl -w
2
3   # Determining whether a given string is a palindrome.
4   # (Recursive approach)
5
6   sub isPalindrome($);
7
8   print "Enter a string > ";
9   chomp(my $str = <STDIN>);
10  if ($str ne '') {
11      print "$str is ", isPalindrome($str)?"":"not ", "a palindrome.\n";
12  } else {
13      print "The string should not be empty!\n";
14  }
15
16  sub isPalindrome($) {
17      my $string = $_[0];
18
19      # A standalone character or an empty string are by definition symmetric.
20      # This signifies the deepest recursion stack possible.
21      if (length($string) <= 1) {
22          return 1;
23      }
24      # Here, what we need to do is to examine the first
25      # and last character, and invoke a new isPalindrome
26      # to deduce whether the string in the middle is a palindrome.
27      my $leftchar = lc substr($string, 0, 1, "");
28      my $rightchar = lc substr($string, -1, 1, "");
29      return $leftchar eq $rightchar && isPalindrome($string);
30  }
```

The only part of concern is the subroutine definition of `isPalindrome()`. In the iterative approach, two pointers are maintained which initially point to the first and the last character, respectively. A loop is set up which iterates as the pointers move towards each other. When an unmatched character pair is found the value 0 is returned, which signifies the string is not a palindrome. Finally, when the positions of the left pointer and right pointer are swapped, that implies the entire string has already been scanned through and all character pairs matched (or 0 would have been returned), so we can then conclude the string is a palindrome.

The logic behind the recursive scheme, however, seems to be cleaner and more intuitive. In the iterative approach, a single `isPalindrome()` invocation tackles the whole of the problem. However, the recursive approach suggests to break this problem into multiple levels. At each level, we merely compare the first and last character of the incoming string. The string in the middle is passed to a new invocation of `isPalindrome()` to deduce whether it is a palindrome. In other words, we define that a palindrome is one whose first and last character are identical and the substring in the middle is also a palindrome. If both conditions are satisfied we conclude the string is a palindrome; otherwise, it isn't.

Lines 21-23 in the recursive example handles the case when the incoming string is a single character or an empty string. In recursive schemes one always need to consider the case at which point recursion should stop. Recursion should not be allowed indefinitely (and in fact, you should avoid recursions of many levels, say possibly 500 levels deep because in practice the stack size is limited and you actually create an entry on the call stack (to be described shortly) as you recurse. Exceeding the limit results in a stack overflow error). Note that as recursion proceeds, the first and last character are being taken off of the string before passing to a new `isPalindrome()` invocation. Therefore, there must be a level at which the incoming string is either a single character or empty, depending on the number of characters in the original string specified by the user. Also note that the ordering of the two conditions on line 29 is significant. Because if we find that the border character pair doesn't match, we can already claim the string is not a palindrome without having to test the string in the middle. I used the short-circuiting property of `&&` to achieve this. If you swap the two conditions, then recursion must always have to proceed to the deepest level and the test is only carried out just before you exit from each level, which is just a waste of time.

Depending on the problem nature, recursion may be a better solution compared with an iterative approach. For example, a program which searches through a directory structure (say, search for certain files on the hard disk) is nearly always implemented by a recursive scheme because directory structure is hierarchical, or in other words, nested by its very nature. Because the number of nested levels is not known in advance, an iterative scheme is unlikely to be appropriate.

Recursion generally requires nested subroutine invocation. How is this possible? Subroutine invocation is internally kept track of by a series of stacks. A **stack** is a data structure. Think of a stack as a pile of books lying on your desk. Only two operations are allowed on a stack, either you place a book at the top of the stack, or you take away a book from the top of the stack. You are not allowed to insert into or remove from it any books in the middle. I will now briefly go into the process of subroutine invocation because certain concepts will be revisited in later chapters.

The basic idea is that a **call stack** is maintained by your program. The top of the stack reflects your current environment. When a subroutine is invoked, a new entry is created on top of the stack to represent the new environment. Each environment holds a copy of the lexical variables (those declared with `my`, to be introduced later on) that are specific to its environment. When the current subroutine terminates, so is the environment and the entry is removed from the top of the call stack. The environment previously in effect is reinstated. Therefore, every invocation is independently represented by the corresponding entry on the stack. This is how subroutine invocations are handled in programming languages. Note that this mechanism does not impose any restrictions to nested subroutine invocation. Nested subroutine invocation is just performed in exactly the same way as invocation of another subroutine.

5.3.4 Creating Context-sensitive Subroutines

As we have seen, several builtin functions are context-sensitive. That is, a function call exhibits different behaviours depending on the context around the function call. Subroutines may exhibit this behaviour too, with the `wantarray()` function.

The `wantarray()` function determines the context around a specific subroutine invocation. Consider the following example:

EXAMPLE 5.3 wantarray()

```
1  #!/usr/bin/perl -w
2
3  # wantarray.pl
4  # Demonstrates how wantarray() may be used
5
6  # Context-sensitive searching:
7  # scalar:
8  #   search for first occurrence of substring and
9  #   return offset
10 # list:
11 #   search for occurrence of substring in incoming list and
12 #   return matched strings
13 sub search {
14     my $substr = shift;
15     if (wantarray()) {
16         # remaining parameters are values being searched
17         my @values = @_;
18         my @retval = ();
19         foreach (@values) {
20             my $index = index($_, $substr);
21             push @retval, (($index >= 0)?$_:());
22         }
23         return @retval;
24     } else {
25         # search in incoming scalar string
26         my $value = shift;
27         my $index = index($value, $substr);
28         return ($index >= 0)?$index:undef;
29     }
30 }
31
32 my @items = ('systematic', 'system');
33 my $scalar = 'delinquency';
34
35 my $search1 = search('que', $scalar);
36 print "'que' ", $search1?'':'not ', "found in $scalar\n";
37
38 my @search2 = search('tic', @items);
39 $" = "\n";
40 print "'tic' found in the following items:\n@search2\n";
```

In this example, `search()` is a context-sensitive subroutine which exhibits two different behaviours, depending on scalar or list context. On line 35, `$search1` introduces a scalar context to the rvalue. Therefore, there is a scalar context around `search()`. Whereas on line 38, there is a list context instead. The `wantarray()` function returns a true value in a list context, while `undef` in a scalar context. Many constructs in this example are introduced at a later time in this chapter. However, the idea is pretty simple. In a list context, the subroutine accepts a variable number of parameters which are searched for the existence of the given substring. Strings with contain the given substring are returned as an array. On the other hand, in a scalar context, the subroutine accepts two parameters (you may pass additional parameters, but they are simply ignored) indicating the substring and the

string being searched. The offset associated with the first occurrence of the string is returned using the `index()` function. I will not describe the `index()` function in detail in this tutorial, because its uses can be substituted by regular expressions which I will cover in Chapter 9.

I would like to take this opportunity to introduce to you another predefined variable `$"` which appears on line 39. This is called the **list separator**. As illustrated in the example, substitution also applies to arrays in double-quoted strings. Therefore, you should escape all `@` characters in string literals so that they are not interpreted as array variable substitution. The list separator is embedded in between list elements during array variable substitution of double-quoted strings.

5.4 Packages

When you split your code into multiple files, Perl provides a nice mechanism to ensure that variables in different parts of your program do not clash. The mechanism is to divide your program into different **namespaces**. The idea is very simple — each namespace has a label which uniquely identifies the namespace and we prepend to variable names the label so that we can differentiate in case two variables in two namespaces happen to have the same name. C++ uses the notion of namespace, while in Perl terminology a namespace is called a **package** instead.

Any variables not explicitly contained in any packages belong to the `main` package. Therefore, all variables we have been using in fact belong to the `main` package. By declaring additional packages we create shields so that variables in different packages would not interfere with each other. Packages are fundamental to object-oriented programming because each object is intended to be a self-contained unit.

5.4.1 Declaring a Package

A package extends from the package declaration up to the end of the enclosing code block, the closing bracket of `eval()` (see Chapter 10) or the end-of-file, whichever comes first (see the perlmod manpage). To declare the start of a package, put

`package` *`package_name`*`;`

Usually package declarations are placed at the beginning of source files to ensure that all variables in the file are protected. For example,

```
#!/usr/bin/perl -w

package Apple;

# Package extends to the end of the file
```

If a package declaration is placed inside a code block, the package extends to the end of the code block:

```
#!/usr/bin/perl -w
```

```
# 'main' package
{
    package Apple;

    # package 'Apple' extends to the end of the block
    $var = 3;
}
# 'main' package
```

To avoid any misconceptions that may arise as you read on, I would like to remind you that packages appearing inside code blocks, such as in the example shown above, continue to exist after the block is closed. Always bear in mind that packages are only intended to prevent inadvertent clashing of namespace. It has nothing to do with scoping. In the above example, the variable `$Apple::var` still has the value of 3 after the containing block terminates. This is not the case for `my` or `local` variables, though, which we will come to shortly.

Note that a package may be declared within the extent of another package. As illustrated in the above example, the 'Apple' package is declared within the extent of the 'main' package. Therefore, you can declare an 'Orange' package within the extent of the 'Apple' package, and each package protects the variables within its respective extent. By convention, the first letter of a package name is capitalized, except the 'main' package.

5.4.2 Package Variable Referencing

If you omit the package name when referencing a variable, e.g. `$somevar`, it refers to the variable inside the current package. This also applies to variables in the `main` package. Therefore, by not explicitly declaring any packages in previous chapters we have been referring to variables in the current package, that is, the `main` package.

If you need to refer to a package not contained in the current package, you need to qualify the variable with the package name prepended, with `'::'` as the package separator. Therefore, to refer to a scalar variable `$var` in the `apple` namespace you need to write `$apple::var`. If the package name is empty but contains the package separator, e.g. `$::var`, the `main` package is assumed.

Because you need to explicitly qualify a variable with the package name if you have to refer to it in another package, you will not modify it inadvertently unless that is your intention. That is exactly how namespaces work.

There is also an old syntax of using a quote instead of double colon for referring to package variables, for example, `$orange'var`. This syntax may be deprecated in future versions of Perl. However, because it will be interpolated in double-quoted strings, you should beware of strings such as `"$people's pen"`. You should disambiguate by putting a pair of curly braces around the variable name, such as `"${people}'s pen"`.

5.4.3 Package Variables and Symbol Tables

Each package in Perl maintains its own **symbol table**. A symbol table keeps track of a list of symbols defined in the current package and their memory locations at which they can be found. For non-lexical variables (that is, those not declared with the `my` modifier) Perl needs to keep track

of them because they are not confined to any environments. You may access the symbol table of a package through a hash whose name is the name of the package, followed by two colons. For example, the hash representing symbol table of the `main` package is `%main::`. The keys of the hash are the names of symbols defined in the package. The corresponding values is a scalar representing an internal data structure of Perl known as a **typeglob** which in turn holds the references to the actual symbols.

To help you understand it, consider the `@INC` array that we described earlier in this chapter. This array is one of the predefined variables in Perl that is automatically listed in the symbol table of the `main` package. Therefore, there exists a key 'INC' in `%main::`, that is, `$main::{'INC'}`. The value is a typeglob which holds a list of references of symbols with the name INC, that is, `@INC` and `%INC`. The typeglob is represented by `*main::INC`. The typeglob has a number of slots, each of which stores the references of a different type such as scalar, array, hash, anonymous subroutine, filehandle and typeglob (which is just a reference to itself). If there isn't a symbol of one type, the corresponding slot is simply null. You can access the reference of `@INC` and `%INC` through the symbol table with a so-called `*FOO{THING}` syntax. In this example, that are `*main::INC{ARRAY}` and `*main::INC{HASH}` respectively.

Typeglobs were mainly used for parameter passing in earlier versions of Perl when references were not yet in the Perl language. In the next chapter you will be taught on how to use pass-by-reference for parameter passing. You may wonder why I need to mention typeglobs at all if it is no longer actively used. First, in order for you to understand how `local` or package variables work, you'll need to know what a symbol table is. And, because the entries in a symbol table are represented by typeglobs, it is difficult for me not to mention typeglobs at all.

5.4.4 Package Constructors with `BEGIN` {}

In each package you may define one or more subroutines with the name `BEGIN`, which are automatically executed when the package is being parsed. During parsing, when perl sees these subroutines, they are automatically executed in the order these subroutines are defined. That means, these subroutines are executed before the rest of the program does. Customarily, the `sub` keyword is omitted.

That being said, these subroutines are not truly subroutines. After they have been executed, these subroutines are not registered in the symbol table. Therefore, you cannot manually invoke them during runtime. One of the major uses of `BEGIN` blocks is to modify `@INC` to include additional module search paths, especially because `use` is a compile-time statement (modules are loaded during compile-time as opposed to runtime — see Chapter 7). Therefore, you have to ensure that `@INC` gets modified before any `use` statements:

```
BEGIN {
    @INC = (@INC, './libs');
}

use MyLib;      # defined in ./libs
# ....
```

5.5 Lexical Binding and Dynamic Binding

We have already learned how to define environments in a program by establishing code blocks in Section 5.2.2. Subroutine definition is also placed inside a code block so it also defines an environment in itself. However, I have not yet explained how you can restrict an association to within a certain environment. Recall that all variables we have been using are package variables. Once declared, package variables continue to exist in the symbol table as long as the program is running. Before we go into the details of the two types of bindings with respect to associations, let us first examine the general concept of referencing first.

Let's execute this program on your system:

```
1 for (keys %main::) {
2     print $_, " => " , $main::{$_}, "\n";
3 }
4
5 $abc = 3;
```

The generated lines that are of interest at this point are shown below, with others omitted:

```
...
stdin => *main::stdin
ARGV => *main::ARGV
INC => *main::INC
ENV => *main::ENV
abc => *main::abc
...
```

The above code dumps the content of the symbol table. On the left of the arrow are the names of the symbols, while on the right are the corresponding typeglobs. What appears to be interesting is that an entry for `abc` exists in the symbol table, regardless of the fact that the statement which assigns 3 to `$abc` has not yet been executed at the instant the symbol table dump is made. The reason is that a compilation step, despite invisible to users, was performed before the actual execution which scans the whole program for package variables which are then added to the symbol table. Therefore, before the program is actually executed the symbol table has already been constructed. As noted previously, a symbol table keeps track of the symbols that appear in the program. By doing so, the runtime environment (for example, the perl interpreter) prepares a list of symbols at an early point in time which facilitates it to arrange for storage space in the memory and, most importantly, to prepare for referencing operations that occur during execution of the program.

Whenever a symbol appears in a program, a referencing operation is required to be carried out during execution to deduce which data object is associated with the given symbol. For example, when the statement `$abc = 3;` in the program above was executed, the runtime environment needs to find out which data object is associated with the scalar variable `$abc`. The goal of a referencing operation is to locate this association.

Referencing has been made complicated because of the presence of scopes. Without scopes, referencing is easy because there can be only one variable of a certain name in each package. For example, throughout the duration of a program there can be one and only one scalar variable `$Apple::var` in the `Apple` package. `$Orange::var` is already a different scalar variable and do not interfere with `$Apple::var` at all. Therefore, throughout the program all references to

`$Apple::var` always refer to the same data object. However, this may not apply to those variables which are confined by scoping rules. In Perl, two major types of scoping rules are supported, namely **lexical scoping** and **dynamic scoping**. Both scoping systems base on definition of environments such as code blocks, but the way referencing is performed is different.

Most modern programming languages only support lexical scoping. An association that is lexically scoped is visible from the point in the environment in which it is defined, and all environments that appear inside the extents of that environment, until when another lexically scoped variable with the same name appears in those environments. In Perl, a variable declared with the `my` modifier is lexically scoped. Consider this example:

```
1  my $a = "Hello ";
2  {
3      $a .= "World\n";
4      print $a;       # Hello World
5      my $a = "Bye!\n";
6      print $a;       # Bye!
7  }
8  print $a;           # Hello World
```

When execution proceeds to line 3, Perl needs to find out which data object `$a` refers to. At this instant, the current environment, that is the code block between line 2 and 5, is known as the **local referencing environment**. Perl first finds out that up to this point there is not any lexical variable with the name `a` in the local referencing environment. As local referencing fails, the referencing operation proceeds to search for one in the **nonlocal referencing environments**, by proceeding all the way up through parent environments. Here, we find a lexical `$a` in the parent environment, and the data object associated with that variable is used. Therefore, the string "World\n" is appended to the scalar value held by that data object.

However, on line 5 a new lexical `$a` is declared at this point. Therefore, referencing operation performed at line 6 resolves to this lexical. Note that the lexical in the parent environment is untouched, but cannot be accessed anymore in the current environment. It is described as being **hidden**. When the code block terminates, all local associations are destroyed. Because of the reference-based garbage collection mechanism, the data object associated with the lexical on line 5 is also destroyed. The parent environment that is once invisible springs into existence again, and lexicals that were once hidden are visible again. Therefore, the last `print()` outputs "Hello World".

`my` expects a scalar or a list as its argument. We have seen a scalar used as the argument in the examples. Using a list as an argument with optional assignment looks like this:

```
my ($a, $b) = ('Hello', 'World');
```

If the variable list is not assigned, then they are given the values of `undef`.

Lexical scoping models are recommended for several reasons. First, the use of lexical variables is faster compared with dynamically scoped ones. That is because lexical scoping can be solely determined from the nesting of code blocks, which is already fixed during the compilation phase. Therefore, referencing of lexical variables can be performed during compilation instead of at runtime. Dynamic scoping, on the other hand, also takes into account the dynamic factor of subroutine invocations. In Perl, `local` variables are dynamically scoped. As the use of dynamically scoped variables share similar problems as global variables in other programming languages, and

the scope is dependent on the call stack which is determined by how the program calls subroutines at runtime, they make debugging more difficult, and are relatively slower.

`my` variables are never listed in the symbol table. This fact is important as you go on and learn how to use typeglobs.

Dynamic scoping, on the other hand, is based on the **call stack** instead of nesting of environments. In Perl, `local` variables are dynamically scoped. These variables are package variables and appear in the symbol table of the respective packages. The idea is, when a `local` variable is declared, the current value as can be accessed through the symbol table is saved temporarily in a hidden stack, and a new data object is created to hold the new value. When the current environment terminates, the current symbol table entry is removed, and the value that was previously saved is reinstated. Consider this example:

```
1  sub greeting {
2      print $a;       # Bye!
3  }
4
5  $a = "Hello ";
6  {
7      $a .= "World\n";
8      print $a;       # Hello World
9      local $a = "Bye!\n";
10     &greeting;
11 }
12 print $a;           # Hello World
```

This is similar to the example I used above to introduce `my` variables in Perl, except now `local` is used and the second `print` is put in a subroutine. On line 9, the original symbol table entry for `$a` (with the scalar value "Hello World") is replaced with a newly created data object whose value is "Bye!". In the subroutine `greeting()`, because `$a` cannot be resolved in the local environment, the symbol table entry is used, and therefore "Bye!" is displayed. When the block terminates, the original symbol table entry saved is reinstated. Therefore, the value `printed` is "Hello World".

There are a few points to note here. Here is a slightly modified version of the above program for illustration:

```
1  my $a = "Hello ";
2
3  sub greeting {
4      print $a;           # Hello World!!!!
5  }
6
7  {
8      $a .= "World\n";
9      print $a;
10     local $::a = "Bye!\n";
11     &greeting;
12 }
13 print $a;
```

When you run this program, you would find that all three `print()` result in the string "Hello World" being displayed. The reason is that during compilation phase all references to $a within the lexical scope have already been associated with the lexical variable. If you swap the positions of the lexical variable declaration and the subroutine, you will find that the `local`ized package variable $a is `print`ed in this case.

Also, as shown on line 10, when you try to `local`ize a variable when a lexical variable of the same name exists, you have to explicitly use the double-colon form to indicate the package symbol table entry. That is because, as explained, $a is tied to the lexical variable and trying to `local`ize a lexical variable is a runtime error.

Apart from a package variable, you can `local`ize a member of composite type. For example, you can have

```
local $ENV{'PATH'} = '/home/cbkihong/bin';
```

which causes the original value to be saved and temporarily replaced by the new given value. When the environment terminates the original value is restored.

Because of potential confusions that may arise when you use `local` variables, one is generally not recommended to use `local` variables.

5.6 Conditionals

A programming language is practically not useful if the statements are only allowed to run from the very first line to the last. Therefore, in this section we are going to talk about loops and conditionals.

You have used the comparison and logical operators in the previous chapter. By using conditionals, you can specify a block of code to be executed if a particular condition (test) is satisfied. This is what conditionals exactly do.

The `if-elsif-else` structure is the most basic conditional structure. The general form is:

```
if (EXPR1) BLOCK_1
[elsif (EXPR2) BLOCK_2] ...
[else BLOCK_n]
```

The parts in square brackets denote the optional parts. The `if-elsif-else` structure works as follows: if EXPR1 evaluates to true, statements in BLOCK_1 are executed, and the remaining `elsif` or `else` parts are bypassed. Otherwise, Perl jumps to the next `elsif` or `else` part, if any.

Perl goes to the `elsif` part if the previous condition is not met (i.e. false). If EXPR2 evaluates to true, BLOCK_2 is executed and the remaining parts are bypassed. There can be as many `elsif` parts as you like, and Perl will test each condition successively until any test evaluates to true. The `else` part is placed at last, handling the situation when all the previous tests failed. The BLOCK_n will be executed in this situation.

Figure 5.1 presents the flowchart showing the sequence of actions performed inside an `if-elsif-else` conditional structure.

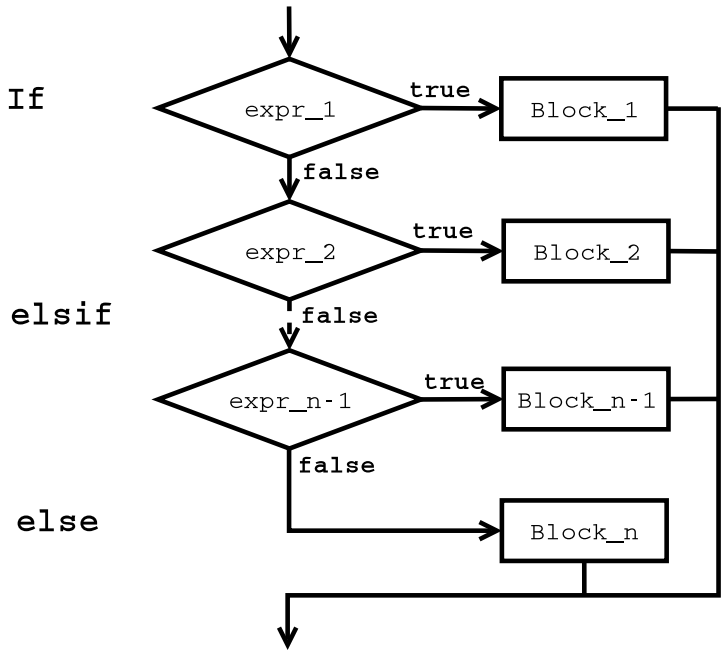

Figure 5.1: If-elsif-else Flowchart

The following is a simple program in which the user inputs a number, and the program deduces whether it is an even number, odd number or zero.

EXAMPLE 5.4

```
1  #!/usr/bin/perl -w
2
3  $numtype = "";
4
5  print "Please enter an integer > ";
6  chomp($num = <STDIN>);
7
8  if ($num % 2 == 1) {
9      $numtype = "an odd number.";
10 } elsif ($num == 0) {
11     $numtype = "zero.";
12 } else {
13     $numtype = "an even number.";
14 }
15
16 print $num . " is " . $numtype . "\n";
```

This program has all the three parts, forming a complete `if-elsif-else` structure. Because 0 is customarily considered neither odd nor even, we have taken this special case into consideration. First it tests if the number is odd by checking if the modulus (remainder) is 1. If this is true, line 9 would be executed. Otherwise, it jumps to line 10 to test if the number is 0. If this test fails again, we know for sure that it should be an even number.

Perl also has an `unless` conditional structure. The following example illustrates its use:

```
1 #!/usr/bin/perl -w
2
3 print "Please enter your age > ";
4 chomp($in = <STDIN>);
5 unless ($in < 18) {
6     print "You are an adult.\n";
7 } else {
8    print "You are less than 18 years old.\n";
9 }
```

If you use `unless`, the sense of the test is reversed. Line 6 is executed if the expression evaluates to `false`. If the expression evaluates to `true`, Perl executes the `else` part. In fact, the `unless` structure is somehow redundant. However, Perl gives you the flexibility to do your job in alternative ways. That is an exemplification of the Perl motto *"There Is More Than One To Do It"*. You can replace line 5 with

```
if (!($in < 18)) {
```

... or even

```
if ($in >= 18) {
```

to achieve the same effect.

5.7 Loops

Sometimes we would like to have a mechanism to execute a sequence of statements repeatedly for a specific number of times or under a particular condition. A loop is the answer. First, I will introduce the `for` loop.

5.7.1 `for` loop

The `for` loop is inherited from C/C++. The general syntax is

```
for ([init-expr]; [cond-expr]; [loop-expr]) BLOCK
```

First, the initial expression `init-expr` is executed. In this part usually a variable would be defined that acts as a counter to keep track of the number of times executed. Then the conditional expression `cond-expr` is evaluated. If the expression evaluates to anything other than `undef`, empty string (“”) or the numeric 0 (i.e. the three scalar values that are defined as `false`), the `BLOCK` is executed. After the `BLOCK` has been executed, the `loop-expr` is evaluated. Then, a new cycle starts, and the `cond-expr` is evaluated again until the `cond-expr` evaluates to false, then the loop terminates.

The process described above could best be visualized using a block diagram as shown in Figure 5.2.

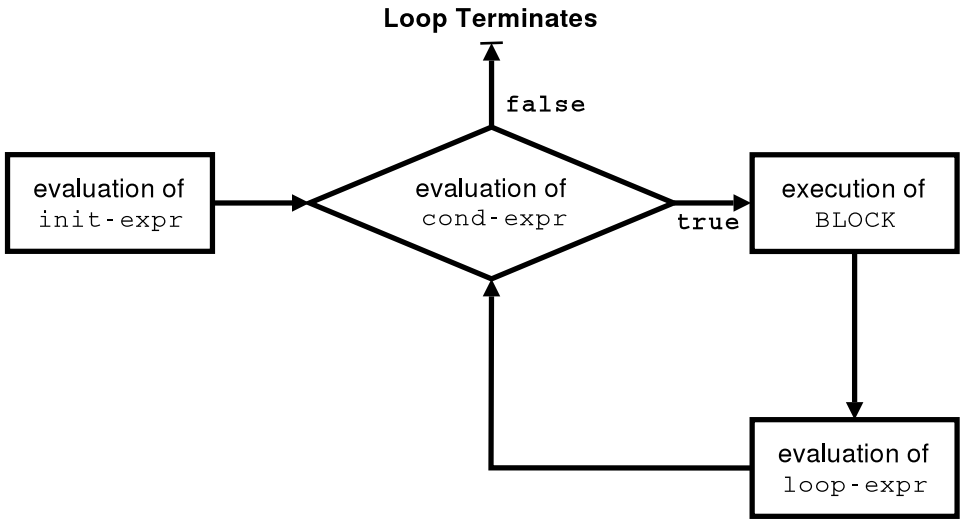

Figure 5.2: for-loop Flowchart

We would now write a script that prints a special pattern on the screen. It consists of two isosceles triangles pointing vertically towards each other.

EXAMPLE 5.5 Double Triangles

```
1  #!/usr/bin/perl -w
2
3  print "Please input the width of the base (1-50) > ";
4  chomp($input = <STDIN>);
5  if ($input < 1 or $input > 50) {
6      die "Input must be in the range (1..50)!\n";
7  }
8
9  for ($trend = 0, $i = $input; $i <= $input; ($trend)?($i+=2):($i-=2)) {
10     if ($i == 1 or $i == 2) {
11         $trend = 1;
12     }
13     print " " x (($input - $i)/2) . "*" x $i . "\n";
14 }
```

In the example, lines 5-7 handles the case if the user enters a number out of the range (1..50). In particular, the `die()` operator on line 6 outputs the error message specified to the standard error (STDERR), which is the screen by default, and then terminate the program.

On lines 9-14 I used the `for` loop to print the asterisks line by line. The `$i` variable stores the number of asterisks to be printed at each cycle, while `$trend` keeps track of whether you decrease or increase the number of asterisks by 2 after each loop. Initially 0 is assigned to `$trend`, so that the conditional operator decrements `$i` by 2 after each loop. The upper triangle is completed when `$i` takes the value of 1 or 2, depending on whether the user has entered an odd or even number. At this point I assign 1 instead, so that in the next loop `$i` is incremented by 2. This loop is repeated until when `$i` is greater than the width of the base specified. In this case, the loop stops.

Note that although line 9 looks a bit complicated and strange, this is grammatically correct. The `init-expr` part consists of two expressions separated with a comma operator. As you have learned

from the previous chapter, both expressions would be evaluated while returning the value of the last expression. However, this return value is actually ignored in this case.

5.7.2 `while` loop

A `for` loop is not the only type of loop structure available. Another form of the loop structure I would like to mention is the `while` loop. This structure is simpler compared with `for` loop, and the syntax of which is as follows:

```
while (cond-expr) BLOCK
```

How does it work? First, `cond-expr` is evaluated. If it evaluates to true, `BLOCK` is executed. After that `cond-expr` is tested again, and the loop just goes on indefinitely until `cond-expr` evaluates to false.

5.7.3 `foreach` loop

Now I would introduce to you another loop structure that works closely with the list data structure. The general syntax of a `foreach` loop is as follows:

```
foreach [[my] $loop_var] ( list ) BLOCK
```

In every cycle of a `foreach` loop, an element from the specified array or list (`list`) is retrieved and assigned to a temporary `local` variable `$loop_var`, and `BLOCK` is executed. Looping continues until all the elements in `list` have been enumerated. For example, if we would like to check if a particular element exists in an array, we can use a `foreach` loop and iteratively checks if the returned element matches the data we are looking for, as in the example below:

```
1  #!/usr/bin/perl -w
2
3  $searchfor = "Schubert";
4  @composers = ("Mozart", "Tchaikovsky", "Beethoven", "Dvorak", "Bach",
5                "Handel", "Haydn", "Brahms", "Schubert", "Chopin");
6  $prompt = "$searchfor is not found!\n";
7  foreach $name (@composers) {
8      if ($name eq $searchfor) {
9          $prompt = "$searchfor is found!\n";
10         last;
11     }
12 }
13 print $prompt;
```

As mentioned previously, we should have used a hash in the first place, but we use a loop to demonstrate the use of `foreach` loop anyway. In this example, each of the names in `@composers` is compared with "Schubert" in turn, the name we are looking for. The loop keeps on going unless the name specified is found in the list, or all the elements have been exhausted without resulting in a match. It is apparent that "Schubert" is in the array, so we must always obtain a positive result.

In each `foreach` cycle, an element from the specified list or array is assigned to the scalar variable specified. If the variable is omitted (note that `$loop_var` is an optional argument), it defaults to

$_. This is a special variable that Perl, in general, assigns temporary data to if no scalar variable is specified in certain operations. This special variable is used quite often in later chapters, like regular expressions, to shorten the length of the script. However, in my opinion, although you are allowed to make your scripts shorter by omitting specifying certain variables in Perl, it may cause your script to look more cryptic than necessary. However, many Perl programmers use such shorthands, and you should know how to interpret them, and this is the reason why I cover this here.

The variable `$loop_var` is in the form of a `local` variable. However, you may want to restrict it to be lexical scoped. In this case, put `my` after `foreach`, as shown in the syntax above.

Some Perl programmers are lazy to type 7 characters for `foreach` so you may use the shorthand `for` instead. Perl can differentiate whether you use the `for` loop or `foreach` loop from the syntax.

Also notice line 10. The `last` statement is one of the **loop control statements**. A loop control statement controls the execution of the loop. In the next subsection we would explore the loop control statements available in Perl.

You may attach an optional `continue` block after a loop block, such as `while` or `foreach`, which will be executed before starting another iteration to evaluate the conditional expression. This is similar in purpose to `loop-expr` in the `for` loop, except this is a block instead of an expression. This is handy if you have to preform a number of operations before starting a new iteration that is otherwise clumsy or cannot be represented by an expression. Here is the double triangle example earlier rewritten with a `continue` block:

EXAMPLE 5.6 Double Triangles (with continue block)

```
1  #!/usr/bin/perl -w
2
3  print "Please input the width of the base (1-50) > ";
4  chomp(my $input = <STDIN>);
5  if ($input < 1 or $input > 50) {
6      die "Input must be in the range (1..50)!\n";
7  }
8
9  {
10     my ($trend, $i) = (0, $input);
11     while ($i <= $input) {
12         if ($i == 1 or $i == 2) {
13             $trend = 1;
14         }
15         print " " x (($input - $i)/2) . "*" x $i . "\n";
16     } continue {
17         $i += ($trend? 2 : -2);
18     }
19 }
```

5.7.4 Loop Control Statements

Loop control statements can only be used inside loops to control the flow of execution.

The `next` statement causes the rest of the code block to be bypassed and starts the next loop iteration. If a `continue` block is present, it is also executed before starting another iteration.

- ⋆ For `for` loops, `loop-expr` is evaluated, and then `cond-expr` is evaluated;
- ⋆ For `while` loops, `cond-expr` is evaluated;
- ⋆ For `foreach` loops, the next element is taken from `list`.

The `last` statement causes the rest of the code block to be bypassed and the loop then terminates. Execution starts at the statement immediately following the `BLOCK`. Any `continue` block is also bypassed.

5.8 Leftovers

Before we end this chapter, we will look at some miscellaneous issues which have not yet been covered. A `do` block contains a sequence of statements which are sequentially executed, while returning the value resulting from the last statement. A `do` block is usually used with a modifier. Perl has defined a number of **modifiers** which can be attached to the end of a statement. As an example:

```
$i = 0;
do {
    print $i++, "\n";
} while ($i <= 30);
```

which `prints` all the integers from 0 up to 30. If a modifier is attached to a `do` block, the block is executed once before evaluating the condition. If the condition is true, then the block is executed again. Otherwise, execution resumes on the next statement. The possible modifiers are:

```
if EXPR
unless EXPR     # same as "if (!EXPR)"
while EXPR
until EXPR      # same as "while (!EXPR)"
foreach EXPR
```

If a modifier is not attached to a `do` block, as in the following example, the expression in the modifier is evaluated before executing the rest of the statement:

```
$i = 0;
print $i++, "\n" while ($i <= 30);  # prints 0 to 30
```

This is actually equivalent to

```
$i = 0;
while ($i <= 30) {
    print $i++, "\n";
}
```

Summary

- The `require()` function is used to source in external files.
- The `@INC` array stores path prefixes Perl uses to find an external source file.
- An association exists between a variable name and the underlying data object.
- The scope of an association refers to the region in which the association is visible.
- Scopes are determined by nesting of environments. An environment is established inside a pair of curly braces.
- A subroutine consists of a sequence of statements defining an environment, which may accept a list of scalars as parameters and return a list of values to the caller.
  - Subroutines enforce reusability of code snippets and make debugging easier.
  - Parameters are merged into a single list and passed into a subroutine which may be retrieved from the array `@_`.
  - A subroutine may return a scalar value or a list of values to the caller by using the `return` function.
  - Prototypes allow type checking of subroutine invocations at compile time.
  - In a subroutine you may determine the context around the subroutine invocation by using the `wantarray()` function.
- Recursion refers to the technique of tackling a problem by defining the solution in terms of itself. In programming languages this is achieved by nested subroutine invocation.
- Package, or namespace, is a mechanism to minimize variable name clashes.
  - Variables of the same name may exist concurrently in each package.
  - Variables not within the extents of any packages belong to the `main` package.
  - Package variables are referenced by qualifying the variable name with the package name, followed by `::`.
    - Defaults to the `main` package if `::` is present but the package name is not specified.
    - Defaults to the current package in effect if both the package name and `::` are absent.
  - Package variables are resolved from the symbol table.
  - Each package may have multiple `BEGIN` blocks which are executed at compile time.
- An association that is lexically scoped is visible from the point in the environment where it is established and all environments that exist within the extents of the current environment, until the end of the current environment. `my` variables are lexically scoped.
- The scope of an association that is dynamically scoped depends on the call stack. A dynamically scoped association exists as long as the current environment is not destroyed. `local` variables are dynamically scoped.

Web Links

5.1 Coping with Scoping
http://perl.plover.com/FAQs/Namespaces.html

Questions

A. In Example 3.7 I presented a binary search program to check if a user-specified integer exists in the list of 100 randomly-generated integers. Introduce the following changes to the program:

- ⋆ convert it from iterative to recursive approach;
- ⋆ only the search routine is sufficient. You are not required to write the parts which accept user input and generate integers in random;
- ⋆ use lexical variables wherever appropriate;
- ⋆ the search routine should be placed inside a package with name `Search`, in an external file "binarysearch_recursive.pl";

There should be at least a `binarySearch()` subroutine which is in the `Search` package (namespace), which can be invoked as follows:

```
require 'binarysearch_recursive.pl';
my $exists = Search::binarySearch(@array, $integer);
```

where `@array` is any array containing a list of integers, `$integer` is the integer to search for, and `$exists` is 1 if the integer is found, `undef` otherwise.

You may save the number of comparisons taken to reach a conclusion as a package variable `$numSteps` in the `Search` package that can be accessed directly or through a subroutine.

Chapter 6

References

6.1 Introduction

We have covered the use of scalars in the first few chapters of this tutorial. However, we haven't covered a very important type of scalar yet — **references**. Perl references are prevalently used nowadays, thanks to the official support of object-oriented programming starting from Perl 5. You need to have a good command of references before heading towards the topic of object-oriented programming.

References look cryptic from the start. It resembles the concept of "pointers" in C/C++, and "hard links" in Unix filesystems (you may flip to Appendix D for a crash course on basic Unix). The use of references is in some sense a peculiar concept in programming. However, it turns out to be very useful in parameter passing and serves as the basis for constructing complex data structures in Perl (and therefore, it is central to object-oriented programming in Perl). Towards the end you will also learn to use typeglobs in your programs. Typeglobs were used to pass data structures to and from subroutines when references were not yet available in earlier versions of Perl. Today, they have been largely superseded by references, and the section on typeglobs is merely presented for your information only.

A reference is a special form of scalar variable that stores the location (address) of a data structure. Fundamentally, when something needs to be stored in the memory, a memory address is required. It is only with the knowledge of the address of the object that we can access it. This is synonymous with your postal address in mail delivery, or your IP address for packet routing on the Internet.

6.2 Creating a Reference

To create a reference, prefix the \ operator to the data object. For example,

```
$a = \100;
```

This creates a reference variable `$a` that points to a newly created data object which holds the literal 100. If the memory address of the data object is stored is at location `0x8101B8C`, the reference `$a` would have an rvalue of `0x8101B8C`. The following diagram is a pictorial representation of the situation. Note that the memory address varies from system to system, and because of the relocatable nature of processes, the address may probably vary on every execution. I just made it up here for the purpose of illustration.

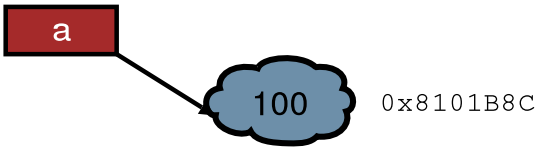

Figure 6.1: Scalar Reference

In the diagram, the association is represented by a solid arrow. The real data object is represented by a cloud shape while the reference by a rectangle. You can further create a reference $b which points to $a, by

```
$b = \$a;               # scalar reference
```

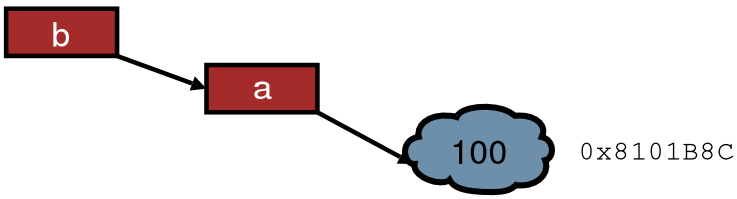

Figure 6.2: A Chain of Scalar References

Apart from references of scalars, you can also create references of hashes, arrays, subroutines and typeglobs. Subroutine reference (or code reference/anonymous subroutines) and typeglob reference will be revisited afterwards.

```
$arrayref = \@array;        # array reference
$hashref = \%hash;          # hash reference
$coderef = \&subroutine;    # subroutine reference
$globref = \*typeglob;      # typeglob reference
```

Though appearing awkward to do so, you may create a chain of references in one go without involving intermediate variables by prefixing multiple backslashes to the data object. For example, the above two statements are functionally equivalent to:

```
$b = \\100;
```

Be careful of enumerated lists as a special case! As indicated on the perlref manpage, taking a reference of an enumerated list evaluates to a list of references of the list elements:

```
@list = \($a, $b, $c);      # Actually (\$a, \$b, \$c)
@list2 = \($a, @b);         # Actually (\$a, \$b[0], \$b[1], ...)
# Actually \$c. Remember list operator in scalar context?
$scalarref = \($a, $b, $c);
```

Recall that in an enumerated list all arrays or hashes are expanded to the constituent elements to form a single list. To create a reference to an enumerated list directly without creating an intermediate array first, which is called an **anonymous array**, enclose the list in [ ] instead:

```
$listref = [$a, $b, $c];
```

Similarly, to construct an **anonymous hash**, enclose the key-value pairs in { } instead:

```
$hashref = {
    'key1' => 'value1',
    'key2' => 'value2',
    'key3' => 'value3',
};
```

Because a reference is simply a scalar value, elements of an anonymous array (or values of anonymous hash) can also be a hash reference or array reference. In this way, we will be able to implement some complex data structures easily, which we shall explore later in this chapter as well as in the next chapter when I introduce object-oriented programming. For example,

```
$hashref = {
    'values' => [
        'a',
        'b',
        'c',
    ],
    'device' => 'screen',
    'options' => {
        'indent' => TRUE,
        'color' => '0xFFFF00',
    },
};
```

which creates an anonymous hash containing three key-value pairs. The `values` key maps to an anonymous array, while the `options` key maps to an anonymous hash. The `device` key maps to a simple scalar string.

References to subroutines behave like functors in C. They can be created in two ways. The first way is to prepend `\&` to the name of the subroutine. The `&` is required when you are referring to the name of a subroutine. For example, if you have a subroutine named `somesub()` you can take a reference to it by `\&somesub`.

You can also create an anonymous subroutine directly, by using `sub` without specifying the name of the subroutine:

```
$subref = sub {
    # This is a subroutine
};
```

6.3 Using References

If you try to `print()` a reference variable, the type as well as the memory location will be displayed, for example `CODE(0x814f3a0)` for an anonymous subroutine.

Once you have the reference variables, you may **dereference** them to access the underlying data objects. To dereference a reference variable, simply put a pair of curly braces around the reference variable and prepend it with the symbol which stands for the underlying type. For example, to dereference the reference variable $a in our earlier examples, we can write ${$a}. Other examples:

```
@array = @{$arrayref};          # array
$scalar = ${$arrayref}[0];      # Return the first element of array above
%hash = %{$hashref};            # hash
$scalar = ${$hashref}{'KEY'};   # Return the value whose key is 'KEY' of hash above
&{$coderef}('a', 'b');          # subroutine invocation
```

In general, you can omit the curly braces around the reference variables in dereferencing operations. Situations that require them will be described at a later time.

You can also dereference multiple levels deep. For example, consider the chain of references $b we saw at the beginning of this chapter, we can access the data object which holds the literal 100 by dereferencing it two times, by

```
$a = \100;
$b = \$a;
print $$$b;         # prints 100
```

However, creating a reference to a literal makes it read-only. For example, trying to modify its value in dereferencing is a runtime error:

```
$a = \100;
$b = \$a;
$$$b = 90;      # same as $$a = 90

Modification of a read-only value attempted at test.pl line 3.
```

Because the value pointed to by $a is read-only, we can only change it by creating a new literal of the desired value, and point $a to it instead. $b would now reflect the new value when dereferenced:

```
print "Before reference \$a changes: $$$b\n";
$a = \200;
print "After reference \$a changes: $$$b\n";
```

The output is

```
Before reference $a changes: 100
After reference $a changes: 200
```

This is different from the case below, which you may change the underlying value because because the reference is taken on a scalar variable, not a literal:

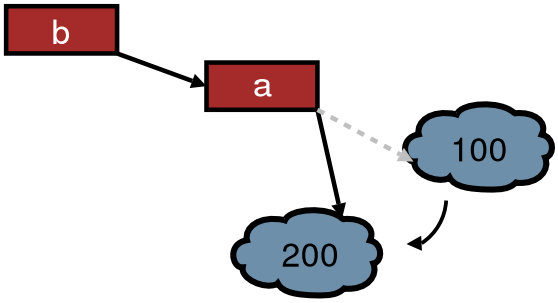

Figure 6.3: Change in Reference Chain

```
$a = 100;
$b = \$a;
$$b = 90;
```

Note that subroutine prototypes are ignored when you invoke a subroutine through its reference, because prototypes are only used during compile time, at which point perl cannot yet resolve the subroutine to which the reference variable points.

Recall that to dereference a reference variable we put a pair of curly braces around it and prepend a backslash to it. I then told you that the braces are customarily omitted. The fact is, the curly braces may contain anything, provided it returns a reference matching the expected type. For example,

```
${ $$ref{'KEY'} }
```

dereferences the hash reference `$ref`, take the value associated with key KEY which is presumably a scalar reference, and dereference it. Note that this is not the same as `$$$ref{'KEY'}`, which is actually identical to

```
${${$ref}}{'KEY'}       # $$ref->{'KEY'}  (see below)
```

which dereferences the scalar reference `$ref` to get the underlying hash reference, dereference it and then access the value associated with the key KEY. In other words, when the curly braces are omitted, access of array or hash is performed at last. As the reference chain gets more complicated the dereferencing expression can get very confusing. Therefore, Perl also supports a special syntax resembling the C pointer-to-member (arrow) operator. Table 6.1 summarizes the alternative forms.

Operation	Alternative Method
`$$arrayref[$index]`	`$arrayref->[$index]`
`$$hashref{$key}`	`$hashref->{$key}`
`&$coderef(@args)`	`$coderef->(@args)`

Table 6.1: Alternative Syntax for Dereferencing

The arrow operator can be cascaded, for example,

```
$hashref->{'files'}->[0] = 'index.html';
```

presumably `$hashref` is a hash reference, and `$hashref->{'files'}` yields an array reference. However, Perl is smart in that by executing the above statement it will automatically create all the necessary data structures and references to fulfill this statement if they do not yet exist. This is known as **autovivification**. In other words, Perl essentially executes the following statement when `$hashref` doesn't exist:

```
$hashref = {
    'files' => [
        'index.html',
    ],
};
```

6.4 Pass By Reference

Recall that Perl combines the input parameters into a single list. Similarly is the case when a subroutine returns a list of scalars to the caller. Therefore, you cannot pass multiple lists to a subroutine as you cannot separate them into respective lists. References are useful in this regard because, no matter how complicated the underlying data object is, they are simply represented by a scalar, and you can pass scalars easily as usual. Moreover, passing by reference — even if you copy it to another variable — allows you to access and possibly modify the underlying data object. Pass by reference is efficient because only a scalar value is involved in parameter passing. When used with subroutine prototypes, subroutine invocation can never be more flexible than ever.

Passing a reference is easy. Simply take the reference, and pass it as a parameter and get it back from `@_` in the subroutine. With prototypes (and it is not invoked using the `&` form) the caller does not even need to take the reference himself/herself if the corresponding symbol in the prototype is prefixed with the backslash `\` symbol. The following example demonstrates both methods:

```
sub passArgs (\%$) {
    my ($hashref1, $hashref2) = @_;
    # Both variables are now hash references
}

%myhash = ('key1' => 'value1', 'key2' => 'value2');
passArgs(%myhash, \%myhash);
```

However, because prototypes may not be observed, depending on the way the subroutine is invoked, you may wish to require the caller to take the reference and pass the reference instead of relying on prototypes, which is in my opinion the safest way to go at present.

Perl does not support named parameter passing, but some programmers may also wish to emulate it as follows, which is recommended if you have to pass a number of parameters to a subroutine.

```
sub somesub (%) {
    my %params = shift;
    # All parameters now in %params.
    my $filename = $params{'FILE'};
    my @args = @{$params{'ARGS'}};
```

```
}

somesub(
    'FILE'  =>  '/bin/ls',
    'ARGS'  =>  [
        '-l',
        '-R',
        '/home/cbkihong',
    ],
);
```

The advantage is you don't need to remember and follow a predetermined ordering of parameters because there is no ordering in a hash.

6.5 How Everything Fits Together

Here, let us consolidate the concepts you have learned in this and the previous chapter. This is also an appropriate opportunity to form a more complete picture as to how they all fit together.

By now you should already have a clear separation of the names and their underlying data objects. Data objects exist independently from variable names as seen in programs. They are linked together by an association, or in other words, a binding. Bindings of lexical variables are determined and fixed in the compilation phase, before a program is even executed. Which data object a lexical variable is bound to is solely determined by the nesting of environments. Bindings of dynamic variables, including those declared with the `local` modifier are resolved at runtime from the symbol table, and are dependent on the call stack during program execution.

Creating references to a data object actually implies establishing additional access paths to it. In general, you can access a data object in two ways. The first way is to access it through a variable which is visible and is bound to the data object. The second way is to access it through dereferencing a reference variable which points to the data object. Perl maintains the number of associations to every data object to determine when they should be freed. Simply put, Perl would destroy a data object when the number of associations to a data object drops to 0. This indicates it cannot be accessed from within the program anymore, and can therefore be safely destroyed. Consider this example:

```
sub createArray () {
    my @array = (1, 2, 3);
    return \@array;
}

{
    my $arrayref = createArray();
    print scalar @$arrayref, "\n";
}
# $arrayref and underlying object destroyed
```

In the `createArray()` subroutine, a data structure in the form of an array is created which holds the three elements, and a lexical array variable `@array` is created that associates with the array. Before

the return function is executed, a reference to the array is created, which adds an association to the array. By the time the return function has been executed, the lexical association to the array has been destroyed. However, because a reference to the array still exists, the array is not deallocated. Therefore, when the number of elements of the array as accessible through the lexical array reference is printed on line 8 the value displayed is still 3. However, at the end of the block the lexical array reference is destroyed, so the reference count to the array is now 0, and the array object is also destroyed.

6.6 Typeglobs

In the previous chapter we had a brief introduction to typeglobs. A typeglob represents an entry on the symbol table and stores a list of references to data objects with the same name. In this section, we describe how you may use typeglobs in your programs. In many cases, you seldom have to mess with typeglobs anymore as references are more convenient and flexible.

Package variables, including those declared with local are resolved from symbol tables. You can use typeglobs to create symbol table aliases. For example, *array = *myarray; causes referencing operations on the symbol array to be resolved through the symbol table entry of myarray instead.

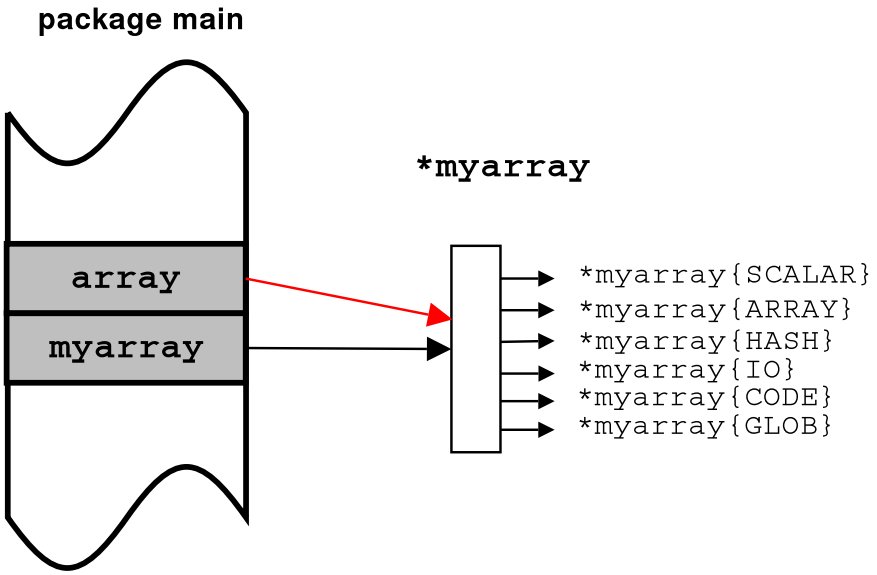

Figure 6.4: Symbol Table Aliasing

This causes $array to refer to $myarray, @array to @myarray and &array to &myarray etc. However, you rarely would like to create an alias for all types in practice. You can actually assign a reference of the desired type to a typeglob to replace the existing one. In Section 5.3.2 we wrote an insert() subroutine that allows you to insert a list into any given array at any arbitrary position. You may also rewrite it in this way to make use of typeglobs:

```
sub insert (\@$@) {
    local *myarray = shift;     # not my!!
    my ($pos, @list) = @_;
    return splice(@myarray, $pos, 0, @list);
```

```
}
```

Note that, unlike references, accessing through a typeglob does not require any dereferencing. With a typeglob entry in place you simply replace the * symbol with whatever type symbol that is required for the type. Some people may consider it convenient as a result.

Also note that you cannot use a lexical variable, that is one declared with `my`, to represent a typeglob because a typeglob is a symbol table entry which by nature cannot be lexical. However, this does not prevent you from assigning a lexical reference variable to a typeglob, though.

Finally I would introduce to you the `*foo{THING}` notation, which you have already seen in the previous chapter in passing using `*INC` as an example. By using this notation you can get the individual references held in a given typeglob `foo`. Replace `THING` with the name of the type. The table below lists the possible types and their names:

THING	Type
SCALAR	Scalar reference (\$foo)
ARRAY	Array Reference (\@foo)
HASH	Hash reference (\%foo)
CODE	Subroutine reference (\&foo)
GLOB	Typeglob reference (*foo)
FORMAT	Format Variables (Not covered in this tutorial)
IO	Filehandle

The notations in parentheses represent an alternative way of accessing them, using references. Because of the absence of alternative ways to get the reference to a filehandle, `*foo{IO}` is most probably the only one that is more widely used. This is used when a file handle needs to be passed to and from a subroutine. We will talk about filehandles and the generic input/output mechanism at a later time.

While typeglobs are still useful in modern Perl programming, you are generally advised to use them only if you have good reasons to do so, especially if that purpose can be fulfilled with other means such as references. First, typeglobs cannot be lexical variables while reference variables can. Also, typeglobs cannot be used to create complex data structures that are possible with references.

Now you should have the fundamental knowledge to proceed to object-oriented programming.

Summary

- Reference is a scalar value whose value is the address of a data structure.
- A reference is created by prefixing the \ symbol to a data object.
- An anonymous array is an array reference, which can be constructed by a pair of square brackets `[]`.
- An anonymous hash is a hash reference, which can be constructed by a pair of curly braces {}.
- An anonymous subroutine is a subroutine reference, constructed by a subroutine definition without specifying a name.

- A data structure of arbitrary complexity can be easily implemented in Perl by using hash and array references as building blocks.
- You may access the underlying data object through a reference by dereferencing it.
- Pass by reference refers to the practice of passing a reference of a data structure as argument to a subroutine, which is the best way to pass an array or hash to a subroutine while not having its elements merged into `@_`.
- Named parameter passing may be emulated by passing all name-value pairs through a hash.
- Creating references to a data object actually implies establishment of additional access paths to it. Perl maintains a reference count for all data objects and they are not destroyed until the reference count drops to 0.
- A typeglob represents an entry of the symbol table and stores a list of references to data objects with the same name.
- References stored in a typeglob may be accessed using the `*FOO{THING}` syntax.

Chapter 7

Object-Oriented Programming

7.1 Introduction

Object-oriented programming (OOP) is a popular term in the programming community. It represents an alternative approach of programming to cope with program development in the large. As I outlined in the previous chapter, maintenance of large scale programming projects become increasingly difficult as the size of code base increases. Object-oriented programming is seen by many in the Software Engineering community as a desirable solution to keep complex programming projects in order. In the text below we would explore the rationale behind this argument and point out how object-oriented model can help alleviate some of the deficiencies in the plain old procedural model.

Perl started to support the notion of object-oriented programming in Perl 5. In fact, most of the necessary concepts that you need to understand object-oriented programming in Perl has already been covered in the previous chapters, and frankly, not much content are left for this chapter. However, because of the importance of object-oriented programming, and in order to avoid making the previous chapters too long if I lump them together, I have to dedicate a chapter to object-oriented programming. Compared with previous chapters, you will find longer and more complete examples in this chapter, so as to help you familiarize with OOP in a more practical context. You should flip back to the previous chapters if you have not read them carefully, because the Perl implementation of OOP is based on subjects studied in the previous chapters. In other words, knowledge of packages, scope, subroutines and references is a prerequisite to understanding OOP in Perl.

If you have previously programmed in some other object-oriented programming languages like Java and C++, you may safely skip the section "Object-oriented Concepts" below. However, because the Perl implementation of OOP is very much different from other languages like PHP, Java and C++, you should not skim read the remaining sections as you would find the Perl approach to object-oriented programming alien to you (it happened to me as well when I learned Perl with some knowledge in C++), and there are some traps to which programmers who have written object-oriented programs in other languages are vulnerable. Please keep this in mind when you are reading this chapter.

7.2 Object-Oriented Concepts

7.2.1 Programming Paradigms

The syntax of a programming language is largely influenced by the **programming paradigm** on which the language is based. A programming paradigm represents a framework which describes in a general, language-independent way how syntactic language elements are organized and processed. Without delving deep into a formal definition of programming paradigms, which is far beyond the scope of this tutorial, I would rather state in a more casual way that a programming paradigm is supported by a school of thoughts to address a specific subset of programming tasks. Therefore, it is generally not appropriate to strictly claim that one paradigm is always better than the other. However, I personally believe if properly implemented, a properly-written object-oriented program is easier to maintain, and allows for a larger basis of extension by leveraging the power of object-oriented programming, for reasons that I would explain later in this section.

There are various programming paradigms in use today. However, the majority of programmers adopt either one of two major paradigms, namely procedural programming and object-oriented programming.

Our discussion so far has been solely based on the procedural programming paradigm. A complicated program is broken down into smaller pieces by delegating pieces of the source code to subroutines. and external source files. Variables are created and updated directly to maintain program state during execution.

Object-oriented programming should not be considered a total revamp of procedural programming. Instead, it is best considered one that uses procedural programming as the basis with the emphasis on the way different logical components (i.e. classes) interact with each other. It enforces a more well-defined way of grouping related subroutines and data into a logical entity, and this is the foundation of object-oriented programming.

For an executive summary of the various programming paradigms, please visit the Wikipedia.org entry on programming paradigm. Be prepared, they are conceptual computer science topics that cannot be easily understood.

7.2.2 Basic Ideas

As I mentioned in the previous section, the idea of object-oriented programming is to group related subroutines and data into logical entities, each of which constituting its own domain. Such logical entities are known as **classes**. Each class defines a framework which describes the set of properties and behaviours of **objects** created (**instantiated**) from the class.

Consider an airplane and a car. Because a car and an airplane have different characteristics and exhibit different behaviours (e.g. fly vs. move) we model them using two different classes. In other words, each class represents a certain type of object. In a class, behaviours are implemented as **methods**, which are subroutines associated with a class. For example, an airplane class would very likely include an `ascend()` method and a `descend()` method which contain the program needed for escalation and landing of an airplane. Also, different classes are likely to have different **properties**. For example, an airplane is likely to maintain an `altitude` property to track down the current height of the airplane above the ground. On the other hand, a car would not have this property. A class only defines the properties. The values of which are maintained independently in

each class instance.

After we have established a class, we need to create a class instance which is called an object. When a class instance (an object) is created, it possesses all the methods of the class and, as noted in the previous paragraph, each object holds an instance of the properties. This arrangement allows each object to carry its own set of property values. For example, consider a `Car` class which has only one property `colour` and one method `move()`. When we create several objects from the `Car` class, each of them "inherits" the method from the class and maintains a value representing the colour of the object.

7.2.3 Fundamental Elements of Object-Oriented Programming

An object-oriented programming language needs to qualify three fundamental properties, namely **encapsulation**, **inheritance** and **polymorphism**. These principles are fundamental to support the virtues of object-oriented programming.

Adapted from Wikipedia.org, encapsulation refers to *the practice of hiding data structures which represent the internal state of an object from access except through public methods of that object*. Basically, that implies you should not change a property by directly modifying the internals of an object. Instead, you should modify it through the **interface** of the object. An interface is what is expected to be seen from outside of the object. In Perl, this includes all object methods. For example, while an airplane object maintains the `altitude` property, you should not modify its value directly. Instead, you invoke the methods `ascend()` and `descend()` to change its altitude because some actions need to be taken before climbing up or going down, which are accounted for by the two methods. By interacting with methods through the interface, the methods can check whether the operation is valid before committing any changes to the object.

Inheritance allows a class (subclass) to inherit methods from another class (superclass). For example, a `Helicopter` class may inherit from the `Airplane` class the `ascend()` and `descend()` methods, while adding a `stationary()` method which an airplane doesn't have and is a characteristic of a helicopter. By inheriting from another class, you don't need to write the inherited code again (unless you would like to override them). Inheritance allows creation of code that can be easily reused.

Polymorphism is a more abstract notion that cannot be easily explained without resorting to examples. The principle is that it allows programmers to use a consistent interface for method invocation on objects of different classes.

These concepts would be revisited later on. However, before we go further, let us write a simple Perl program with object-oriented perspective so that you can appreciate how an object-oriented program looks like in Perl.

7.3 OOP Primer: Statistics

In this section we write a simple Perl module `Stats.pm` which calculates the mean, variance and standard deviation of a set of input numbers. We then write a perl program `stats.pl` which uses the module written to output these statistics given the input of a set of numbers. The program

listing is given first, and the theories are revisited afterwards.

EXAMPLE 7.1 Statistics Calculator

```
# Stats.pm
# The "Stats" Perl module

package Stats;

# Create a new class instance (object)
# and return a reference of the object
sub new {
    my $arg0 = $_[0];
    my $cls = ref($arg0) || $arg0;
    my $this = {};
    bless $this, $cls;
    $this->clear();
    return $this;
}

sub clear {
    my $this = $_[0];
    $this->{'numlist'} = undef;
    $this->{'x_sum'} = 0;
    $this->{'x2_sum'} = 0;
}

# Append a value to the list
sub addValue {
    my $this = $_[0];
    my $num = $_[1];
    if (defined $num) {
        push @{$this->{'numlist'}}, $num;
        $this->{'x_sum'} += $num;
        $this->{'x2_sum'} += $num**2;
    }
}

# Calculate total
sub getTotal {
    my $this = $_[0];
    return $this->{'x_sum'};
}

# Calculate mean
sub getMean {
    my $this = $_[0];
    my @numlist = @{$this->{'numlist'}};
    if (!@numlist) { return 0; }
    return $this->getTotal()/@numlist;
}
```

```
48
49  # Calculate variance
50  sub getVariance {
51      my $this = $_[0];
52      my @numlist = @{$this->{'numlist'}};
53      my $n = @numlist;
54      my $sum_x2 = $this->{'x2_sum'};
55      my $sum_x = $this->{'x_sum'};
56      if (!$n) { return 0; }
57      return ($n*$sum_x2 - $sum_x**2)/($n**2);
58  }
59
60  # Calculate standard deviation
61  sub getStdDev {
62      my $this = $_[0];
63      return $this->getVariance()**0.5;
64  }
65
66  # Get list of values
67  sub getValueList {
68      my $this = $_[0];
69      return @{$this->{'numlist'}};
70  }
71
72  1;
```

```
1   #!/usr/bin/perl -w
2
3   # stats.pl
4   # This program uses Stats.pm to print out some assorted
5   # statistics on the input numbers
6
7   use Stats;
8
9   # Catch Ctrl-C (SIGINT signal)
10  $SIG{'INT'} = 'getResults';
11
12  my $obj = new Stats;
13
14  sub getResults {
15      print "\n\nResults ============================\n";
16      print "Number of values: ", scalar($obj->getValueList()), "\n";
17      print "Total: ", $obj->getTotal(), "\n";
18      print "Mean: ", $obj->getMean(), "\n";
19      print "Standard Deviation: ", $obj->getStdDev(), "\n";
20      print "Variance: ", $obj->getVariance(), "\n";
21      exit (0);
22  }
23
24  print qq~
25  Statistics Calculator
```

```
26  Calculates several sets of statistics given a sequence of input numbers.
27
28  Enter one value on each line.
29  To exit, press Ctrl-C.
30  ~;
31
32  while (1) {
33      print ">> ";
34      chomp(my $num = <STDIN>);
35      $obj->addValue($num);
36  }
```

When the program is executed, the output looks like this:

```
1   cbkihong@cbkihong:~/docs/perltut/src/oop$ perl -w stats.pl
2
3   Statistics Calculator
4   Calculates several sets of statistics given a sequence of input numbers.
5
6   Enter one value on each line.
7   To exit, press Ctrl-C.
8   >> 13
9   >> 26
10  >>
11
12  Results ============================
13  Number of values: 2
14  Total: 39
15  Mean: 19.5
16  Standard Deviation: 6.5
17  Variance: 42.25
```

In this transcript, 13 and 26 were input and then Ctrl-C was pressed to signal the end of input list. The results are then displayed. This program catches (intercepts) the SIGINT signal, which is generated when you press Ctrl-C. By default, if you press Ctrl-C when a program is running it promptly terminates the program. In this program, I demonstrated how to install a **signal handler**, which is a subroutine that is automatically invoked when the corresponding signal is caught. The signal handler is executed instead of terminating the program.

Signals are messages sent by the operating system to a process (a program in execution). Because signal is a notion from Unix, and the message broadcasting mechanism is different on every operating system, behaviour of signal handling is platform-specific. Signals are readily supported on Unix platforms by using the signal handling system calls. Windows has limited support of signals. Support for common signals like SIGINT seems to be working on Windows, though. The example program works on both of my testing platforms, namely GNU/Linux and Activestate Perl 5.8.0 on Windows. I chose to use signal because catching of the SIGINT signal and executing the `getResults()` subroutine is handled automatically so we don't need to place any extra code in the `while` loop to detect the end of input list. However, because an explanation of signals is out of the scope of this tutorial, and is not the main theme of this chapter, my discussion of signals will stop here.

7.3.1 Creating and Using A Perl Class

Perl does not have any specialized syntax for classes as in many other object-oriented programming languages like C++, Java and PHP. That makes OOP in Perl looks more cryptic than it really is. A Perl class is contained in a file of extension '.pm' in its own package. This is called a **module**. The filename of the module is the name of the class followed by '.pm'. The package name is also the name of the class. Recall from the previous chapter that if the package name contains `'::'`, it is changed to the directory separator as the pathname when the module is being sourced. The following table displays some example package names and the corresponding locations where they need to be saved.

Package Name	File Path (relative to `@INC`)
`Stats`	Stats.pm
`Crypt::CBC`	Crypt/CBC.pm

Table 7.1: Relationship of Package Names and File Placements

Recall that `@INC` contains a list of path prefixes that Perl uses to locate a Perl source file. A module contains all method definitions. Don't forget to put the `1;` at the end of the module. Omitting this results in a compile-time error.

Before you use a module you should first import it into your program. You can use `require` that you were taught in Chapter 5. You may import a module in one of two ways. The first way is to pass the path to the module (relative to `@INC`) in quotes to the `require` function. For example, `require "Crypt/CBC.pm";` Another way is to just specify the package name without quotes, for example `require Crypt::CBC;` However, the `use` function is preferred in general for Perl modules. The syntax of `use` is

```
use MODULE [LIST];
```

It is semantically equivalent to

```
BEGIN {
    require MODULE;
    import MODULE LIST;     # indirect object syntax
}
```

which not only imports the module `MODULE` at compile time, it also imports the symbols specified by `LIST` into the current namespace (package) using the `import` class method. `import` is a special class method that may be defined by a class author. Many modules do not have any symbols to be exported, however, some of them do. If the method cannot be found, it is silently bypassed. For example, users of the `CGI::Carp` module may import from it the `fatalsToBrowser` symbol, which is actually the name of a subroutine that generates a page in HTML describing the details when an error occurs, which is then returned to the viewer's browser. This is used by CGI scripts to easily trap runtime errors. The `use` statement required is

```
use CGI::Carp 'fatalsToBrowser';
```

Documentation of a class should tell you how to `use` it, and whether any symbols need to be imported. Therefore, usually you don't have to worry about this at all, unless you have to write a

class that exports symbols. The use function is generally preferred to require because modules are imported at compile time. Therefore, missing modules are discovered at the time the program is compiled before execution, which saves you from an embarrassing situation where execution is on midway when Perl finds out some of the modules are missing and the program fails to continue.

7.3.2 How A Class Is Instantiated

As mentioned previously, to instantiate a class is to create an object that belongs to the class. From the perlobj manpage, a Perl object created is *simply a reference that happens to know which class it belongs to*. How does everything fit together?

Let us go back to the new() class method of the Stats class in our example. The most important statement is on line 12, that is, the bless() statement. The bless() statement makes the reference $this no longer just an ordinary hash reference. It becomes an object reference of the Stats class, and by doing so you can also access the methods from the reference. Therefore, this method is known as a **constructor**, because it is where the object is created. A constructor is not necessarily called 'new()'. It can be of any name, despite it is customarily called new. Object-specific data, or properties, are stored into the object reference. Let us explain each line one by one.

```
my $arg0 = $_[0];
my $cls = ref($arg0) || $arg0;
```

In Perl, methods are exactly identical in behaviour to subroutines. However, whether it acts like a method or a plain subroutine is determined solely by the way it is invoked. A method is a subroutine that expects an object reference as its first argument. Using the new() method as an example, the following ways of invoking it are identical:

```
my $obj = new Stats;            # Indirect object syntax
my $obj = Stats->new();         # C++-like class member invocation (recommended)
my $obj = Stats::new(Stats);    # Resembling subroutine invocation
```

With the indirect object syntax, the name of the method is placed first, followed by the object reference or name of the class. For example, getTotal $obj ('a', 'b') means that the method getTotal() is invoked on the object $obj. Any arguments that follow ('a', 'b') are arguments to the method. This form is generally only recommended for constructors, because this form of method invocation is not syntactically obvious. The second form is generally recommended as it's less confusing. The class name or object reference is placed first, followed by ->, the method name and arguments. The last form is typical subroutine invocation. However, in this form you need to pass the class name or object reference as the first argument.

Now go back to our example. We put the first argument to $arg0. The next line serves to obtain the class name from it. The ref() function tests if the parameter is a reference. If it is one, it returns a string indicating its type (see the ref() documentation for details). If it is an object reference, it returns the package name. By the short-circuiting behaviour of logical operators the package name is assigned to $cls. This caters for the fact that the new() method can be invoked on the Stats class or a class instance, although customarily, a constructor is only invoked on a class in most other programming languages.

```
my $this = {};
bless $this, $cls;
```

Here, we create an empty hash reference and `bless()` it to the `Stats` class. This operation is called `bless`, very likely because it works like a wizard playing with his magic wand, turning an ordinary reference into an object. Very magical indeed. To be an object, Perl only mandates a `bless()`ed reference. It doesn't require to be a hash reference, although it is most likely chosen for its flexibility. Please read the perltoot manpage for its coverage on alternative forms of Perl objects. It accepts two parameters. If the second parameter is missing, it assumes the current package.

The last two lines in the constructor resets the object data and return the object reference so that the caller of the constructor (in stats.pl as `$obj`) can save the object reference for use in later operations.

The program reads in a number from each line, and invoke the `addValue()` method on `$obj`. This invokes the method with the object reference itself as the first parameter. The value, as the second parameter, is appended to a list of numbers that is stored with the object reference, and update the sum (`x_sum`) and the sum of squares (`x2_sum`). These two pieces of statistics are used to calculate the various statistics, when the SIGINT signal is detected and the `getResults()` signal handler invoked. The variance and standard deviation are given by

$$\text{Standard deviation } (\sigma) = \frac{1}{n}\sqrt{(n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2)}$$
$$\text{Variance} = \sigma^2$$

The class provides a number of methods for users of the class to retrieve the list of values, total, mean, variance and standard deviation. Note that these methods are invoked on the `blessed` object returned by the constructor instead of on the class. Through the class interface, the class provides all the necessary methods to interact with without requiring the user to know anything about the implementation of the class. For example, you do not have to care about how data are stored and processed internally. Because an object is simply represented by a reference variable, unlike some other programming languages Perl does not have the notion of protected access specifiers to classify methods by different access permissions such as "public" or "private". Users may also be possible to manipulate the underlying hash directly. However, by documenting the interface clearly class users will be encouraged to access the object through the interface instead of having to manipulate object data directly from the reference variable. This fulfills the initiative of encapsulation.

7.4 Inheritance

Inheritance is important in any programming languages because it allows you to reuse portions of another source program in your program, so that you don't have to rewrite them. You save development effort as a result. Perl implements inheritance in a very simple way. When a method is invoked on an object and the method cannot be found in the current package, the packages whose names are in the variable `@ISA` are searched in order for the missing method. In object-oriented terminology, we describe this situation as a **derived class** inheriting from a **base class**, because the derived class contains methods defined in itself as well as from the base classes. A

base class is also called a **superclass** or a **generic class**, while a derived class is also called a **subclass**.

When you put the name of a package into your package's `@ISA`, all the methods defined in that package, together with any defined in their base classes as well as the base classes of base classes etc. are available in your package. You may wish to redefine some of the inherited methods because they may not suit your current class anymore. To redefine a method, which is called **overriding** a method, simply rewrite the method in the way you desire and put it in the current package. If the method is invoked on the current package or its subclasses, the overriding method will be invoked instead of the overridden ones (that is, those in the base classes).

As an example, we will write a `Stats2` class which inherits from `Stats` with the added functionality of deducing the maximum and minimum among the list of numbers input. Also attached below is the modified `stats2.pl` which uses `Stats2` with the additional functionalities:

EXAMPLE 7.2 Statistics Calculator (Extended)

```
# Stats2.pm
# Derived "Stats" with some functionalities added

package Stats2;

@ISA = ('Stats');   # inherits Stats
use Stats;

# Note that by using the 2-argument form of
# bless() the constructor can also be inherited

# Override addValue()
sub addValue {
    my ($this, $num) = @_;
    if (!defined $this->{'min'}) {
        $this->{'min'} = $num;
        $this->{'max'} = $num;
    } else {
        $this->{'min'} = ($num < $this->{'min'})?$num:$this->{'min'};
        $this->{'max'} = ($num > $this->{'max'})?$num:$this->{'max'};
    }
    # invoke the base class version of addValue()
    $this->SUPER::addValue($num);   # OR $this->Stats::addValue($num)
}

# Find minimum
sub getMinimum {
    my $this = $_[0];
    return $this->{'min'};
}

# Find maximum
sub getMaximum {
    my $this = $_[0];
    return $this->{'max'};
```

```
}

1;
```

```
#!/usr/bin/perl -w

# stats2.pl
# This program uses Stats2.pm to print out some assorted
# statistics on the input numbers

use Stats2;

# Catch Ctrl-C (SIGINT signal)
$SIG{'INT'} = 'getResults';

my $obj = new Stats2;

sub getResults {
    print "\n\nResults ============================\n";
    print "Number of values: ", scalar($obj->getValueList()), "\n";
    print "Minimum: ", $obj->getMinimum(), "\n";
    print "Maximum: ", $obj->getMaximum(), "\n";
    print "Total: ", $obj->getTtotal(), "\n";
    print "Mean: ", $obj->getMean(), "\n";
    print "Standard Deviation: ", $obj->getStdDev(), "\n";
    print "Variance: ", $obj->getVariance(), "\n";
    exit (0);
}

print qq~
Statistics Calculator
Calculates several sets of statistics given a sequence of input numbers.

Enter one value on each line.
To exit, press Ctrl-C.
~;

while (1) {
    print ">> ";
    chomp(my $num = <STDIN>);
    $obj->addValue($num);
}
```

Note that by placing the name of a package in `@ISA` does not free you from the need of adding the `use` statement to import the contents of the package. `Stats` is the base class while `Stats2` is the derived class. Here, the `addValue()` method is overridden to check whether the current input number is the minimum or maximum and update the counters internally if it is one to always keep track of the minimum and the maximum. Then, two new methods are defined that let users retrieve the minimum and maximum values.

Occasionally an overriding method needs to invoke an inherited but overridden method.

addValue() here is an example, because we overrode it to add new functionalities rather than to replace it in its entirety. You can prefix a package name before the name of the method, separated by two colons, to indicate the class from which to start searching for the method. The default is always to start searching from the package to which an object belongs. Therefore, you always have access to the current, overriding version. You can access the version defined in the Stats package, that is the base class with respect to Stats2, by

```
$this->Stats::addValue($num);
```

However, for convenience a pseudo class SUPER has been defined that always refer to the base classes. Essentially, the current, overriding version is bypassed, as indicated in the example program.

The Stats2 class can actually make use of the getValueList() method to deduce the maximum and the minimum values on-the-fly instead of having to memorize two additional pieces of information, and the addValue() method does not need to be overridden as a result. This is left as an exercise for the readers.

Even if your class's @ISA is empty, it still inherits from a class called UNIVERSAL, which is the base class of all classes in Perl. Therefore, methods defined in UNIVERSAL are available in any class. It has three methods, which are described below.

isa(CLASS)

This method allows you to check whether an object belongs to CLASS or a subclass of CLASS. It returns a true value if it is one, undef otherwise. You may use it in two ways.

Object-oriented form:

```
$obj->isa('UNIVERSAL');     # must be true, by definition
```

Procedural form:

```
use UNIVERSAL 'isa';
isa($obj, 'UNIVERSAL');
```

can(METHOD)

This method allows you to check whether an object has a method called METHOD. It returns the reference to the subroutine if there is one, undef otherwise. It also searches the upstream base classes. As an example, this is an indirect and silly way to invoke the isa method on the current package:

```
sub isSubclass {
    my ($this, $cls) = @_;
    my $subref = $this->can('isa');
    $this->$subref($cls);
}

$obj->isSubclass('UNIVERSAL');
```

Can you add methods to a package like UNIVERSAL without modifying the module file? The answer is you can. Here, I am going to demonstrate how to add to the UNIVERSAL class a constructor so that you don't have to define any constructors in your modules.

EXAMPLE 7.3

```
# universal.pl
# Contains definitions to add to UNIVERSAL class

package UNIVERSAL;

sub new {
    my $cls = ref($_[0]) || $_[0];
    my $this = bless {}, $cls;
    $this->initialize();
    return $this;
}

sub initialize {
    # defaults to no-op
    # override to execute class-specific startup code
}
1;
```

```
# TestObject.pm
# Test class module

package TestObject;

sub initialize {
    print "TestObject::initialize()\n";
    $this->{'a'} = 6;
}

sub displayMessage {
    my ($this, $msg) = @_;
    print $msg;
}

1;
```

```
#!/usr/bin/perl -w

# testuniversal.pl

BEGIN {
    # Add additional methods to UNIVERSAL
    require "universal.pl";
}

use TestObject;

```

```
12 my $obj = new TestObject;
13 $obj->displayMessage("Hello World!\n");
```

`universal.pl` contains the methods to add to the `UNIVERSAL` class. This file is sourced by a `BEGIN` block in `testuniversal.pl` which ensures that the module definitions are imported at compile time. Note that the `TestObject` module does not have to define the constructors as a result. Note that you may optionally specify an `initialize()` method in your class module, which is automatically invoked by the constructor (`universal.pl`::9) where you can place some initialization code. If you do not provide it, the version provided by `universal.pl` will be used instead, which is just an empty method that does not do anything.

Although you may do it that does not necessarily mean you should do it. In general, I suggest converting `universal.pl` into a module which acts as the base class for all your other modules instead of adding new methods to the `UNIVERSAL` class. This example is given to demonstrate it works. However, if a user of `TestObject` forgets to import the `universal.pl`, then it will be an error as a constructor cannot be found.

Summary

- Object-oriented programming refers to the practice of grouping related subroutines and data into a self-contained logical entity, called a class.
- Each class defines a framework which describes the set of behaviours and properties of objects instantiated from the class.
- Methods are subroutines associated with the class. Properties are attributes possessed by objects instantiated from the class.
- Encapsulation refers to the practice of hiding internal class implementation from users. Class users are expected to only interact with it through public methods.
- Inheritance allows a subclass to inherit methods from a superclass, so that you may save development effort through reuse.
- A class module in Perl is simply a package, whose file extension is '.pm'.
- The `use()` function is usually used to load a Perl module at compile time.
- Every Perl class has a constructor which is responsible for creating objects belonging to the class.
- The `bless()` function associates a reference variable with a class in order for the reference to become an object.
- A method inherited from a superclass may be overridden by specifying the definition for the subclass.
- In class methods the `SUPER` pseudo class may be used to invoke an overridden method.
- All classes inherit a class named `UNIVERSAL`.

Web Links

7.1 perlboot manpage — Object-Oriented Tutorial for Beginners
http://www.perldoc.com/perl5.8.0/pod/perlboot.html

7.2 perltoot manpage — Tom's Object-Oriented Tutorial for Perl
http://www.perldoc.com/perl5.8.0/pod/perltoot.html

7.3 perltooc manpage — Tom's Object-Oriented Tutorial for Class Data in Perl
http://www.perldoc.com/perl5.8.0/pod/perltooc.html

7.4 perlbot manpage — Object-Oriented Bag of Tricks
http://www.perldoc.com/perl5.8.0/pod/perlbot.html

Chapter 8

Files and Filehandles

8.1 Introduction

The ability to read from and write to files is nearly always essential to computer programs. Output is frequently generated in the course of execution of a program. However, if they are not stored in secondary storage media such as disks, they will disappear once the power is switched off. Therefore, file access is an important element of the input/output system. In this chapter, we will explore the general Perl input/output system and the functions we can use to access the filesystem.

Similar to the C standard and to be in line with Unix concepts, Perl uses the concept of **filehandles** to represent an opened file. They are also known as **file descriptors** in programming languages like C. Although I did not mention explicitly, actually you have been working with filehandles throughout the tutorial. The statement

```
print "Hello World!";
```

Appeared in the very first Perl program covered in this tutorial, it implicitly uses a filehandle to get the string `print`ed on the screen. This statement is actually written as

```
print STDOUT "Hello World!";
```

However, the filehandle `STDOUT` is assumed by default, so we have been omitting it all the way up to this chapter. Unix has a generalized view of input and output. Apart from files, hardware devices are also represented as files on a Unix filesystem. Therefore, input and output of hardware devices can be represented by input and output of files, through a filehandle. This generalized view helps abstract a user or programmer from the peculiarities of the implementation of each device, which are left to the device drivers that exist in the operating system kernel, allowing users to interact with each device in a uniform manner. With a strong Unix tradition, Perl adopts a similar notion as well. Even if you don't use any flavours of Unix at all it does not mean you don't have to read this chapter, because Perl uses the same concepts on every platform it runs on, so that a considerable degree of platform independence can be achieved.

8.2 Filehandles

Perl has a number of predefined filehandles, namely `STDIN`, `STDOUT` and `STDERR` that you can use in command line applications for input/output on screen. `STDIN`, or standard input, is the filehandle

from which keyboard input can be read. STDOUT, or standard output, is where you should send the output of your program. STDERR, or standard error, is mostly used for outputting warning or error messages because sometimes people wouldn't like error messages to be displayed. By dumping error messages to a separate filehandle the user may decide whether to display them or instead redirect them to, for example, a log file on disk for diagnosis at a later time. These three filehandles are always ready for you to use and you don't need to create them manually.

Filehandle is one of the data types available in Perl. As you have learnt in the chapter on references, filehandle has a separate slot in a symbol table entry. However, unlike other data types you do not have to prefix a filehandle with a type symbol. By convention, filehandles are in all capital characters to make them visually stand out from names of functions and subroutines etc.

8.2.1 open a File

Unless you are working on one of the predefined filehandles, the first step you should take is to populate a filehandle. A populated filehandle represents an active **stream** which allows input or/and output of data. For the case of file access, a populated filehandle represents an opened file, which is then used by the input/output functions to read from or write to the file. Filehandle is also used in Perl socket programming which is used to represent a socket. However, due to time constraints network programming is not covered in the first edition of this tutorial.

Unix supports two sets of file access functions. One set is provided by the operating system kernel, and the other set is provided by the standard C libraries installed on the system, which is implemented on top of the version provided by the kernel. The file access functions in Perl are actually interfaces to these functions. Perl allows access to both, through the open() function which invokes the version provided by the C libraries, and the sysopen() function invokes the operating system version. The use of open() is generally preferred to sysopen(), because it is simpler to use, but sysopen() is more powerful. sysopen() is not covered in this edition of the tutorial.

You use the open() function to open a file. Usually, the open() function takes on one of the following forms:

```
open FILEHANDLE, EXPR
open FILEHANDLE, MODE, EXPR
```

FILEHANDLE is either a filehandle or a lexical variable with the undef value, which is used by the open() function to store the reference to the filehandle created. EXPR is a scalar expression which contains the name of file to open(), and MODE describes the access mode to apply to the file, for example, whether read or write access are allowed on the file. If MODE is missing, it defaults to "<", the read-only mode. Otherwise, MODE should be prepended to EXPR in the first form, if it is not given as a standalone argument. If open() is successful, it returns a nonzero value. Otherwise, undef is returned. You should always check the return value of file access functions and handle cases of failure to ensure your program is fault tolerant, especially if your program is to be used by other people instead of you, such as CGI scripts.

Traditionally, a filehandle is usually used instead of a lexical filehandle reference. For example,

```
my $retval = open(LOG, "<command.log");
```

which creates LOG permanently on the symbol table of your current package. This may be acceptable to you, but you may wish to localize it to a certain subroutine, for example. As a recapitulation, this is the way to do it:

```
local *LOG;
my $retval = open(LOG, "<command.log");
```

However, because filehandles themselves cannot be lexical, and many Perl programmers are not familiar with the use of typeglobs, a better way would be to use a lexical filehandle reference, which you can easily pass around without needing any knowledge in typeglobs. An example is

```
my $fhLOG;
my $retval = open($fhLOG, "<command.log");
```

Here is a summary of the 6 access modes provided:

MODE	Description
<	Read-only access. Specified file must exist.
+<	Read-write access. Specified file must exist.
>	Write-only access. File emptied if exists; created otherwise.
+>	Read-write access. File emptied if exists; created otherwise.
>>	Append-only access (file pointer at end-of-file). File created if not exist.
+>>	Read-Append access (file pointer at end-of-file). File created if not exist.

Table 8.1: File Access Modes for open( )

These access modes determine the operations that can be applied on a file. If you open( ) a file with read-only access but you try to write data to the corresponding filehandle, a runtime error will occur.

Every open file has a **file pointer**, which determines the position of the next character read or write. The first four modes listed above position a file pointer at the beginning of a file, so that read/writes occur at the beginning of the file. < grants only reads access to the file. > grants only write access to the file, which is automatically created if it does not exist, and empties it before writing. Both +< and +> grants read-write access to the file, so you may read from as well as write to the filehandle. The difference between +< and +> is that for +<, the specified file must exist, while for +> the file is automatically created if it does not exist, and empties the content before writing. The last two modes place the file pointer at the end of a file. Therefore, data are written at the end of the file. A file opened with either of these two modes is created where necessary.

Note that +< does in-place writing. If file write occurs in the middle of a file, it simply overwrites the exact number of characters from the file pointer required by the write, growing the file as necessary and other characters in the file are unaffected. The effect is similar to putting your text editor in "replace" or "overwrite" mode and typing characters at a cursor to overwrite the old text.

8.2.2 Output Redirection

Output redirection allows you to redirect output sent to a certain filehandle to another filehandle. This is frequently used by shell script authors on Unix systems to redirect error messages to log files

or simply to throw them away as if they have not been output at all.

To use I/O redirection, specify ">&" as the MODE, and EXPR is the name of the filehandle to which output to FILEHANDLE is redirected. An example is shown below, which redirects output that are sent to STDERR to a file that has been previously opened with the filehandle LOG instead.

```
open (STDERR, ">&LOG");
```

Text that are sent to STDERR will be redirected to a file instead, so they are no longer output on screen.

8.3 File Input and Output Functions

In this section, I will introduce to you various functions you can use to read from or write to a filehandle. Note that in functions expecting a filehandle as its argument you can use a lexical filehandle reference instead of a typeglob. Simply replace the filehandle with the lexical variable, for example, $fhLOG.

8.3.1 readline() — Read A Line from Filehandle

The readline() function accepts a typeglob as parameter to read a line from the filehandle contained in the typeglob. In scalar context, each invocation of readline() reads up to the newline character. When no more lines can be read, undef is returned. An example is shown below, which copies a text file File1.txt to File2.txt.

```
open FILE1, "<File1.txt" or die "Cannot open File1.txt!";
open FILE2, ">File2.txt" or die "Cannot open File2.txt!";
my $line;
while ($line = readline(FILE1)) {
    print FILE2 $line;
}
close FILE1;
close FILE2;
```

However, customarily readline() is not frequently used because Perl provides an operator <FILEHANDLE> which is an interface of readline(). We can replace readline(FILE1) above with <FILE1>.

In list context, both readline() and <FILEHANDLE> read all the way until end-of-file occurs, and split it into lines. The return value is an array with its elements being the lines extracted. This is seldom used in practice, because if the incoming file is very large, the generated array will also be very large, consuming a lot of memory space. Therefore, it is a lot safer to set up a loop to read in line by line as shown above.

8.3.2 `binmode()` — Binary Mode Declaration

Not knowing if you are aware or not, text files actually are stored differently on Unix and MS-DOS/Windows systems. The culprit is that the line termination characters used on these platforms are different. That's why there is an ASCII/Binary transfer mode option in your FTP application. Binary files do not rely on line termination characters to denote the end-of-line. Therefore, a binary file is represented in an identical manner on these platforms. However, because text files uses line termination characters, and they vary from platform to platform, two files with identical textual content end up being represented differently on different systems.

To ensure Perl programs have high levels of portability, in general Perl programmers simply have to treat `\n` as the line termination character. This is what we have readily assumed from the very beginning of this tutorial, and the underlying system carries out all necessary conversions for us automatically. However, this system does not work for MS-DOS/Windows systems because they distinguish between text files and binary files. Therefore, on these systems Perl needs to introduce a PerlIO layer on top of the native file access functions which converts between the underlying line termination characters and `\n`. This does not pose any problems for text files, as this conversion is actually intentional for text files. However, binary files should never be altered in any way. `binmode()` with just a single parameter of `FILEHANDLE` essentially instructs all read/write through the filehandle to bypass the conversion layer. Because `binmode()` is ignored on other systems, you should generally use it on all binary files for portability. It should be invoked right after a file is opened, i.e.

```
open BMP, "<logo.bmp";
binmode BMP;
```

If `binmode()` has two parameters, the second parameter indicates the PerlIO layers to apply which act as conversion filters. The "crlf" layer is the layer we mentioned above for MS-DOS compatible systems which carries out line termination conversions. Usage of this form is generally not needed as the defaults are generally adequate for most programming purposes, and so are not described here.

8.3.3 `read()` — Read A Specified Number of Characters from Filehandle

The syntax of `read()` is

```
read FILEHANDLE, SCALAR, LENGTH[, OFFSET]
```

which reads from `FILEHANDLE` `LENGTH` characters, usually equivalent to bytes and assign it to the scalar `SCALAR`. If `OFFSET` is given, it specifies the zero-based offset of `SCALAR` from which to start writing.

This is usually used for binary files, but not necessarily. The file copying program shown above should generally not be used because binary files are not line oriented, and it does not use `binmode()` which causes file copying errors on MS-DOS compatible systems. Presumably the correct way is as follows:

```
sub copy ($$) {
    my ($src, $dest) = @_;
    open FILE1, "<$src" or die "Cannot open $src!";
```

```
    open FILE2, ">$dest" or die "Cannot open $dest!";
    binmode FILE1;
    binmode FILE2;
    my ($buffer, $numChars); my $bufferSize = 1024;
    my $size = 0;
    while ($numChars = read(FILE1, $buffer, $bufferSize)) {
        $size += $numChars;
        print FILE2 $buffer;
    }
    close FILE1;
    close FILE2;
    return $size;
}
```

8.3.4 `print()/printf()` — Output To A FileHandle

We have used `print()` quite a lot so it is not worthwhile repeating all the details here again. However, if a filehandle is specified, it outputs to the filehandle. Otherwise, the filehandle defaults to STDOUT, as I mentioned earlier in this chapter.

`printf()` is similar to `print()`. However, it is an exceptionally powerful function that lets you perform varieties of type conversions on the fly. `printf()` is an artifact from the C standard I/O library. In C, generating a string is not a simple affair, because there is no variable interpolation as in Perl and there is not an easy and flexible way of string concatenation. Also, because C is strongly typed and you cannot concatenate a C-style string with other data types, for example, a number, you have to end up performing a lot of type conversions before the desired string can be successfully generated and eventually written to a file descriptor. Therefore, for convenience purpose C provides a set of functions collectively known as the `printf()` family of functions which allows generation of many common forms of string to be completed in one function call.

While this function is very versatile it is also acclaimed as the most complicated function in C. This function works by constructing a concise but generally cryptic format string, which consists of the string with some placeholder fields inserted. These placeholder symbols describe the type conversion operation required for each of the fields to be inserted into the string, which are passed as additional arguments to the `printf()` function. `printf()` is one of the several few C builtin functions which accept a variable number of arguments. The complexity lies completely in constructing the proper format string. While Perl has flexible string interpolation and automatic type conversions, `printf()` is not as important as in C. Moreover, because `printf()` involves additional operations, it is less efficient compared with `print()`. Therefore, you should avoid it if `print()` suffices for the purpose. However, because it is indeed handy for certain kinds of conversion operations, I am going to describe it below. In Perl's implementation, another function `sprintf()` exists. It returns the generated string instead of directing it to a filehandle. Otherwise, it is identical to `printf()`. In fact, `printf()` is internally implemented using `sprintf()`. This is intended to be an introductory description of most frequently used options only. For more information, please read the perlfunc/sprintf manpage.

The following `sprintf()` example, however simple, gives you a taste of what it is like:

```
# "Good Morning. The number is 6."
sprintf("%s. The number is %d.", "Good Morning", 6);
```

The format string is just a string with placeholders inserted. Placeholders start with the % character. Placeholders are replaced by the corresponding arguments given after the format string with the specified type conversion operations performed. Here, because we have 2 placeholders, we have two additional parameters given after the format string. %s indicates the placeholder value is a string, while %d indicates a signed integer. Perl includes an extension to `printf()` functions that you may specify the placeholder values in an arbitrary order. That is, the arguments may be given in a different order than the order the placeholders appear in the format string. However, because this only complicates matters, I shall not discuss this and mandate all arguments to be given in the same order that the placeholders appear in the format string. s and d here indicate the conversion type. Some other conversion types are defined as well. The most frequently used ones are:

Conversion Type	Description
s	String
d	Signed integer
u	Unsigned integer (decimal)
x	Unsigned integer (hexadecimal, lowercase)
X	Unsigned integer (hexadecimal, uppercase)
f	Floating point number

Here are some examples:

```
sprintf('%X', 12);  # C
sprintf('Absolute value: %u', -1);      # 4294967295 (see below)
```

When using the unsigned integer conversion type you have to ensure the value passed as parameter is not negative. This concerns the way an unsigned integer is represented in the underlying C library. All integers are represented with a fixed number of bits. On my system, this is 32 bits. Therefore, for unsigned integers 32 bits can represent all integers from 0 up to $2^{32}-1$, that is 4294967295. Signed integers are also represented with 32 bits. The method of representation of negative integers is known as "2s complement". The range of signed integers becomes -2147483648–2147483647. Negative integers are represented by inverting the bits of the corresponding positive magnitude and adding 1. Therefore, 1 and -1 are represented as follows:

```
 1          00000000 00000000 00000000 00000001
-1          11111111 11111111 11111111 11111111
```

When you pass -1 as argument while the conversion type is u, the bit sequence of -1 is interpreted as if it is an unsigned integer. Note that this bit sequence is identical to 4294967295 in unsigned integer representation. Therefore, the string returned is 4294967295.

Between % and the conversion type character additional options may be applied. We will dicuss some of them below. You may mix and match them, although a few of them are mutually exclusive as otherwise noted.

You may stipulate a prepended space before a numeric conversion if the numeric value is not negative by inserting a space. For example,

```
sprintf('<% d>', 3);    # < 3>
sprintf('<% d>', -3);   # <-3>
```

By replacing the space with a + symbol, the + sign is prepended instead of a space. Note that the space and + are mutually exclusive. If both exists, + is taken. For example,

```
sprintf('<%+d>', 12);       # <+12>
sprintf('<%+f>', 13.6);     # <+13.600000>
sprintf('<% +f>', 13.6);    # <+13.600000>
```

Once an argument has been converted to the string form, by default the corresponding placeholder is substituted by the string without introducing any padding, as we have seen in the above examples. However, we may introduce extra padding by specifying the minimum field width. It specifies the minimum length of the placeholder value string. If this number is larger than the length of the string, padding is added. By default the string is right-aligned so padding appears on the left. However, if the - symbol is prepended to the minimum field width, the string is left-aligned so padding appears on the right. The padding is in terms of spaces. If the minimum field width is smaller than the string length required, then the field width is ignored. For example,

```
sprintf('<%+12f>', 13.6);       # <  +13.600000>
sprintf('<%+-12f>', 13.6);      # <+13.600000  >
sprintf('<%+-5f>', 13.6);       # <+13.600000>
```

If the string is left-aligned (i.e. format string without the - symbol) you may place a 0 anywhere before the field width to instruct zero padding instead of spaces. For example,

```
sprintf('<%08d>', 12345);       # <00012345>
sprintf('<%0+12f>', 13.6);      # <+0013.600000>
sprintf('<%+012f>', 13.6);      # <+0013.600000>
```

As you have seen, floating point numbers have a certain precision. The precision is influenced by the number of decimal places. Subject to fixed storage space of floating point numbers, the precision is not unlimited as in our usual mathematics. You may specify the precision by specifying the number of decimal places, together with a prepending '.'. For example,

```
sprintf('<%0+12.4f>', 13.6);    # <+000013.6000>
sprintf('<%12.4f>', 13.6);      # <     13.6000>
```

If applied on a string, this specifies the maximum number of characters to fit, somehow like the Perl `substr()` function. For example,

```
sprintf('<%.4s>', 'abcde');     # <abcd>
sprintf('<%10.4s>', 'abcde');   # <      abcd>
sprintf('<%10.4s>', 'abc');     # <       abc>
sprintf('<%-10.4s>', 'abcde');  # <abcd      >
```

8.3.5 seek() — Set File Pointer Position

seek FILEHANDLE, POSITION, WHENCE

You may use the seek() function to set the position of file pointer in a file specified by FILEHANDLE. POSITION is a signed integer indicating the new position, relative to the position indicated by WHENCE. WHENCE is an integer which is either 0, 1 or 2 representing the beginning-of-file, the current file pointer position and the end-of-file respectively. However, because hard-coding these integer values is not semantically obvious, we usually use the names of the corresponding constants available in the C standard I/O library instead. These constants are defined in the Perl Fcntl module. You can import the three constants by

```
use Fcntl ':seek';
```

The constants are SEEK_SET, SEEK_CUR and SEEK_END respectively. After you have imported them, you can use them directly in your programs. seek() is frequently used with binary file access. Binary files usually have their data stored in a certain format that allows efficient access of data. To achieve this, some fields are encoded and saved at fixed positions in a binary file, which you may access directly with the seek() function. The new position is calculated as the sum of POSITION and the base position indicated by WHENCE. Here are some examples:

```
seek(FILE, 0, SEEK_SET);    # Jump to beginning of file
seek(FILE, 5, SEEK_CUR);    # Jump 5 bytes forward
seek(FILE, -1, SEEK_END);   # Jump to last byte of file
```

8.3.6 tell() — Return File Pointer Position

The tell() function returns the position of the file pointer, in bytes. This function returns meaningful values only if used on an open()ed file. The position is a zero-based offset from the beginning of the file specified by the filehandle which is passed as the parameter. It is frequently used with seek() to move the file pointer about in a file for reading or writing. This is an example which deduces the size of a file by using seek() and tell():

```
use Fcntl ':seek';

sub getFileSize ($) {
    my $filename = $_[0];
    local *FILE;
    open FILE, "<$filename" or return undef;
    seek(FILE, 0, SEEK_END);
    my $size = tell();      # last filehandle read is FILE
    close FILE;
    return $size;
}

my $filename = $ARGV[0];
print "$filename\t\t", getFileSize($filename), " Bytes \n";
```

However, on a system that supports stat() this can be satisfactorily replaced by

```
(stat($filename))[7]
```

8.3.7 `close()` — Close An opened File

At the end, when you have finished working with a filehandle, you should `close()` it. Simply pass the filehandle as the parameter to `close()`.

8.4 Directory Traversal Functions

In this section, I will introduce to you various functions you can use to traverse the directory structure of your system. Note that you ought to be using `File::Find`, which is easier to use and more flexible. In fact, it uses the functions that we cover below. However, some simple directory traversal operations may be more efficient if you implement them directly.

Description of the functions is presented first. You will find a full example afterwards which uses these functions to build a class which performs file search.

8.4.1 `opendir()` — Open A Directory

```
opendir DIRHANDLE, PATH
```

This function prepares the directory `PATH` for subsequent directory traversal functions. If the directory exists, a true value is returned, and populates `DIRHANDLE` which is a filehandle.

8.4.2 `readdir()` — Read Directory Content

```
readdir DIRHANDLE
```

The `readdir()` function reads the content of the directory referred to by `DIRHANDLE`, which is populated by the `opendir()` function. In scalar context, each `readdir()` invocation returns an entry in the directory. When there is no more entry, `undef` is returned. This is similar to the behaviour of `read()` saw earlier. You may also get all the entries in one go by calling the function in list context, and an array containing all the entries inside will be returned.

8.4.3 `closedir()` — Close A Directory

When you have finished traversal of the directory concerned, you should `closedir()` it.

8.4.4 Example: File Search

In this example, we will write a class that allows users to search for a file recursively in a directory tree whose name matching a pattern specified by the user. The pattern is in the form of a regular expression for convenience purpose. It is by no means versatile as `File::Find`, but it will give you an idea of how to use the directory traversal functions.

EXAMPLE 8.1 File Search

```
# FileSearch.pm
# An Example File Search module

package FileSearch;

# Create a new class instance (object)
sub new {
    my $arg0 = shift;
    my $cls = ref($arg0) || $arg0;
    my $this = {};
    bless $this, $cls;
    $this->initialize(@_);
    return $this;
}

sub initialize {
    my $this = shift;
    my %options = @_;
    foreach (keys %options) {
        $this->{$_} = $options{$_};
    }
}

sub find {
    my ($this, $path) = @_;
    my @matches = ();
    my $pattern = $this->{'PATTERN'};
    local *DIRENT;

    opendir DIRENT, $path or return ();

    # Get a list of entries in this directory,
    # and sort it lexicographically
    my @entries = sort { lc($a) cmp lc($b) } readdir(DIRENT);

    foreach my $fn (@entries) {
        my $entry = "$path/$fn";

        # Recursive search if the entry is a directory,
        # We should ignore . and .. in our search
        if (-d $entry and $fn !~ m/^\.\.?$/) {
            push @matches, $this->find($entry);
        }
        $fn =~ m/$pattern/ and push @matches, $entry;
    }
    closedir DIRENT;
    return @matches;
}

1;
```

The program below uses this module, getting the root of the search as well as the search pattern from user input, and uses the module to get a listing of files found. Finally, the list of files together with the corresponding file sizes are displayed:

```
1  #!/usr/bin/perl -w
2
3  use FileSearch;
4
5  my ($pattern, $dirroot);
6
7  print "Please specify the filename pattern by means of a regular expression:\n¶
       >> ";
8  chomp($pattern = <STDIN>);
9  print "Please specify the root of directory tree to be searched:\n>> ";
10 chomp($dirroot = <STDIN>);
11
12 my $searchobj = new FileSearch('PATTERN' => $pattern);
13 my @result = $searchobj->find($dirroot);
14
15 foreach (@result) {
16     print $_, " (", (stat($_))[7], " bytes)\n";
17 }
```

The transcript of a sample session is shown below. It searches for files whose extension is .pm. Therefore, a list of Perl modules in the directory tree is shown.

```
cbkihong@cbkihong:~/docs/perltut/src/files$ perl search.pl
Please specify the filename pattern by means of a regular expression:
>> ^.*\.pm$
Please specify the root of directory tree to be searched:
>> /home/cbkihong/docs/perltut
/home/cbkihong/docs/perltut/scrap/Number.pm (344 bytes)
/home/cbkihong/docs/perltut/src/ch06/Controller.pm (916 bytes)
/home/cbkihong/docs/perltut/src/ch06/Stats.pm (1264 bytes)
/home/cbkihong/docs/perltut/src/ch06/Stats2.pm (779 bytes)
/home/cbkihong/docs/perltut/src/ch06/Stats3.pm (425 bytes)
/home/cbkihong/docs/perltut/src/ch06/TestObject.pm (198 bytes)
/home/cbkihong/docs/perltut/src/ch06/Timer.pm (1011 bytes)
/home/cbkihong/docs/perltut/src/ch06/TrafficLight.pm (642 bytes)
/home/cbkihong/docs/perltut/src/files/FileSearch.pm (796 bytes)
```

The `chomp()` function removes the trailing newline character from the variable containing an input string if one is present. Note that the argument must be an lvalue because it performs in-place editing of the string instead of returning the modified string.

The `FileSearch::find()` method is a recursive subroutine which takes a single parameter, that is the base directory in which to search for files matching the specified pattern, which is passed to the `FileSearch` object upon instantiation. The search itself is a **depth-first search**. That means at each level, for each directory entry the content of the directory is searched before the next directory is

searched. Note that we have to avoid . and .. in each directory. In Appendix D in my discussion of hard links I explain what these two entries are. They refer to the current and the parent directory, respectively. Because they go up instead of go down the directory tree, we should not delve into them.

This program looks fine. However, this search implementation can cause the search to go into an infinite loop on Unix systems in a certain case. That happens if two **path segments** on a path point to the same inode. First, let us agree on the terminology first. A path like /a/b/c/d consists of four path segments a, b, c and d. Now, let us assume b and d point to the same directory inode (for an exhaustive explanation of inodes, please read Appendix D). Now c is a directory inside b. However, c is also a directory inside d because b and d are actually just two aliases representing the same inode. Therefore, /a/b/c/d/c is still a valid path, and we can find the directory entry d inside, so we have the path /a/b/c/d/c/d. It is obvious that this circularity will continue endlessly, and obviously you will go into an infinite loop as a result. This is exactly the reason why we have to avoid delving into . and .. in the program. Therefore, this problem normally should not occur. It only arises when a hard link is created manually that makes two directories point to the same inode. The core difficulty is that you cannot easily identify this circularity in the program because d appears exactly like any other directory entries. You are hereby asked to think of a way to patch this potential problem. *(hint: use stat())*

8.5 File Test Operators

Perl provides you with a set of file test operators that you can use to test a file against certain properties and return a truth value. Table 8.2 lists the most commonly used operators:

Operator	Description
`-r`	File is readable by effective user or group.
`-w`	File is writable by effective user or group.
`-x`	File is executble by effective user or group.
`-o`	File is owned by effective user.
`-R`	File is readable by real user or group.
`-W`	File is writable by real user or group.
`-X`	File is executble by real user or group.
`-O`	File is owned by real user.
`-e`	File exists.
`-z`	File is empty (zero size).
`-s`	File is not empty, size in bytes as return value.
`-f`	File is a regular file.
`-d`	File is a directory.
`-l`	File is a symbolic link.
`-u`	File has setuid bit set.
`-g`	File has setgid bit set.
`-k`	File has sticky bit set.

Table 8.2: File Test Operators

Note that many of these options are specific to Unix operating systems. `-e`, `-f`, `-d`, `-z` and `-s` can be used on most operating systems. The only parameter is either a filename or, in case the file has already been `open`ed, its filehandle. For example, to ensure a certain regular file (`abc.txt`) exists,

you can issue the following statement:

```
(-e "abc.txt" && -f "abc.txt")
    or die "abc.txt not found or is not regular file!\n";
```

Notice there are two sets of operators to test whether a file is readable/writable or executable. In general, you should use the lowercase version instead of the uppercase version because the lowercase version uses the effective user or group identity, which always reflect the identity of the user or group under which the program is executed. You can get further information on effective and real user and group in Appendix D with respect to setuid or setgid files.

8.6 File Locking

File locking, or locking in general, is just one of the various solutions proposed to deal with problems associated with **resource sharing**. Sharing of resources frequently arises in our everyday life. Driving on a highway, boarding an elevator etc. are all manifestations of utilization of shared resources. In a computer program, you — the developer — have access to disks, peripheral devices etc. within a program that appears to be self-contained, which in turn leads you into thinking your program has exclusive access to these resources. No, you don't. Resource sharing is equally ubiquitous as, if not more than, in your daily life. In your computer system there may be a few hundred programs in execution (**processes**). Many of them are hidden so you may not know they are running. However, with just one Central Processing Unit (CPU) only one of them can be executed at any instant. Therefore, the CPU actually performs a **context switch** at regular, but very short intervals when a currently-running process is suspended from execution by the CPU with its execution state saved, and another process is resumed. This context switch is performed so frequently that to a user there is a perception that all processes are executed concurrently, but they're not. This is already one example of resource sharing, and the solution is to introduce fine-grained **time slicing**. Similarly, all programs that are executing have access to the disks. Therefore, it is not surprising that when a program is accessing a disk it is possible you can find another one that is also trying to access the disk at the same time, too.

File locks are introduced to set temporary restrictions on certain files to limit how they can be shared among different processes. Depending on the nature of an operation, we come up with two types of locks. The first type is a shared lock, and another one is an exclusive lock. With respect to files, read access can be shared by multiple processes because read access does not result in any changes to the state of the shared resource. Therefore, a consistent view of the shared resource can be maintained. Write access, however, should by nature be carried out with exclusive access until the write operation is complete. Another write operation to the file which occurs before the current one is complete should be queued (certain cases may require it be cancelled instead, depending on the nature of the operation). A read access cannot be concurrent with a write access either because the write access will destroy the consistent view expected by the read access while it is still reading (for example, the write operation may be to delete the file altogether, while the read operation is still in its midway). Therefore, the solution is to introduce shared locks for read access, while exclusive locks for write access. Multiple concurrent shared locks are allowed. However, exclusive locks are mutually exclusive.

On most Unix systems as well as platforms in the Microsoft Windows NT family (including Windows 2000, XP and Windows Server 2003 as of this writing) you may use a handy `flock()` function. Please see the "Function Implementations" section of the perlport manpage to see any platform-specific

issues for your platform. Here is an example demonstrating file locking:

```
#!/usr/bin/perl -w

# writer.pl
# Writes into myfile.dat

use Fcntl ':flock';

open(FILE, ">>myfile.dat") or die "Cannot open myfile.dat!\n";

print "[$$] Requesting exclusive write lock for myfile.dat.\n";
flock(FILE, LOCK_EX);
print "[$$] Requested exclusive write lock for myfile.dat.\n";

print "[$$] Writing (appending) to myfile.dat.\n";
# Any write operations should be put inside
foreach ($ARGV[0]..$ARGV[1]) {
    print "[$$] Writing $_\n";
    print FILE "$_ (process id $$)\n";
    sleep 1;        # sleep one second
}

flock(FILE, LOCK_UN);   # release lock
print "[$$] Released exclusive write lock for myfile.dat.\n";
close FILE;
```

```
#!/usr/bin/perl -w

# reader.pl
# Reads from myfile.dat

use Fcntl ':flock';

open(FILE, "<myfile.dat") or die "Cannot open myfile.dat!\n";

print "[$$] Requesting shared read lock for myfile.dat.\n";
flock(FILE, LOCK_SH);
print "[$$] Requested shared read lock for myfile.dat.\n";

print "[$$] Reading from myfile.dat.\n";
while (<FILE>) {
    sleep 1;
    print $_;
}

flock(FILE, LOCK_UN);   # release lock
print "[$$] Released shared read lock for myfile.dat.\n";
close FILE;
```

This example consists of two programs, namely the writer and the reader. The writer program writes

some sample data into a data file, while the reader program reads from the data file previously written data. The writer takes two command-line parameters which determine the values to be written to the data file. It first acquires an exclusive lock. When the lock is acquired, it starts writing the values to the file. When the write operation is complete, the lock is released. The reader works in a similar manner. It first acquires a shared lock, and when one is acquired it starts reading from the file. At the end, the shared lock is released. To try this example, open two or more commnad-line windows on your desktop environment and try different combinations, such as a writer in a window and a reader in another (writer-reader), reader-reader and writer-writer. Take the writer-reader as an example, when you execute the writer program in one window by

```
perl writer.pl 1 15
```

That means 15 lines of records will be written into the data file. You will see that there is a 1-second pause between each write, because of the `sleep()` function. It is inserted after each write or read so that you can have ample time to switch to another window to execute another writer or reader instance. Now switch to another window and start the reader before the writer has finished writing. No command-line parameters are needed for the reader. You will find that the reader stalls while trying to acquire a shared read lock because an exclusive lock is already in place. When the writer completes, the reader will be able to get the shared lock and proceed with reading. If you try the reader-reader combination, you ought to find that concurrent readers are allowed, which agrees to what I told you earlier.

`flock()` accepts two parameters. The first one is the filehandle. The second argument indicates the locking operation required. Just like the case for `seek()`, it is just an integer whose mnemonic constants can be found by importing the `Fcntl` module, by importing the symbol ':flock'. The shared lock is represented by `LOCK_SH` and the exclusive lock by `LOCK_EX`. To release a lock you can specify `LOCK_UN`. You may optionally bitwise-or `LOCK_NB` with either `LOCK_EX` or `LOCK_SH` to indicate non-blocking. As demonstrated in the example, an `flock()` call will be blocked until the lock is acquired. With this optional flag, you may choose to skip if it fails and do something else, or you may retry it later on. The predefined variable $$ refers to the current process ID. It is displayed as well as saved to file to let you easily identify the various processes.

Summary

- A filehandle is a high-level abstraction in order to provide a uniform interface to I/O involving different media such as file access and socket communication.
- There are three predefined file handles: `STDIN`, `STDOUT` and `STDERR`.
  - User input through the keyboard is read from `STDIN`.
  - Output of a program are usually directed to `STDOUT`.
  - Error messages are usually directed to `STDERR`.
- Before performing any operations on a file, you have to `open()` it. You should `close()` it when you are done with it.
- Output redirection allows you to redirect data sent to a certain filehandle to another filehandle.
- The `readline()` function reads a sequence of characters from a given filehandle until a newline is encountered. The <> operator in Perl serves the same purpose.

- `binmode()` ensures binary files are not corrupted on MS-DOS based operating systems.
- `read()` reads a given number of characters from a filehandle.
- Every open file maintains a file pointer, which you may manipulate with the `seek()` function. The file pointer position may also be obtained by the `tell()` function.
- The `print()` and `printf()` functions output a string to the given filehandle. `printf()` performs some transformations on the string beforehand.
- `sprintf()` is identical to `printf()` except the resulting string to be returned instead of written to any filehandles.
- The `readdir()` function traverses a given directory, whose filehandle as been populated by the `opendir()` function. Upon completion `closedir()` is invoked to close it.
- File test operator can be used to test if a file possesses a given property.
- File locking serves to set temporary restrictions on how certain files may be shared among several processes. The `flock()` function is used.

Chapter 9

Regular Expressions

9.1 Introduction

Now we are marching into probably the most exciting chapter in this tutorial. Regular expressions (regexps, or even RE) are what makes Perl an ideal language for "practical extraction and reporting", as the name implies. To give you an idea, let me first give you an overview on how to perform **pattern matching** in Perl.

First, you construct the **regular expression**, which is essentially a sequence of characters describing the **pattern** you would like to match. The term "pattern" may seem a little bit foreign to you, but you may actually have had some experience of it already. For example, in MS-DOS if you would like to list all files with the extension `.txt` (presumably text files), you may issue a command like

```
dir *.txt
```

On Unix-like operating systems, similarly, you can specify

```
ls *.txt
```

The "`*.txt`" here can be described as a pattern as it is the specification used by the operating system (strictly speaking, the shell, i.e. the program in charge of reading commands from you and displaying output) to look for the file entries that have to be displayed. This is a very simple pattern, but Perl provides users with a very powerful set of regular expressions that can be used to specify the patterns, so you can make your search specification more specific. After constructing the regular expression, you can then bind the data to be searched (for example, text files or just a line of user input) to the pattern using the binding operators =$\sim$ or !$\sim$. In this process, you have provided the Perl regexp engine with both the data to search for and the data to be searched. The return value indicates if pattern matching is successful. If it is successful, you may want to store the data temporarily, or in a file, or export the results directly to the standard output.

Regular expressions are used in Perl in a number of ways:

- ⋆ Search for a string that matches a specified pattern, and optionally replacing the pattern found with some other strings
- ⋆ Counting the number of occurrences of a pattern in a string

- ⋆ Split a formatted string (e.g. a date like 02/06/2001) into respective components (i.e. into day, month and year)
- ⋆ Validation of fields received from submitted HTML forms by verifying if a piece of data conforms to a particular format
- ⋆ ... and much more

Regular expressions are not exclusive to Perl only. It is a vital component in Unix and other Unix-like operating systems like Linux to provide users with powerful text search and replace capabilities. You may find that many software on Unix, like `grep` and `awk`, allows users to use regular expression to specify search specifications (although their implementations are slightly different from that of Perl). Although this is still rare in Windows, many new software with regexp capabilities are emerging because many famous Unix applications are now being ported to Windows and other system platforms. Moreover, a large set of Perl regexps are now being adopted in JavaScript and JScript implementations in Netscape and Microsoft Internet Explorer respectively. This is a piece of good news as sophisticated input validation can now be performed directly by the end users' browsers. In this way, invalid data can be detected without sending anything to the server that causes the transmission delays. In C/C++ there are also regular expression libraries that programmers can easily use for adding regexp capabilities to their programs. That explains how useful regular expressions are in programming nowadays.

As you may know there are books dedicating in their entirety to regular expressions in your local bookstores. Therefore, this chapter is by no means a complete coverage of regular expressions in fine detail. However, after you have finished this chapter you should appreciate how flexible and powerful regular expressions are in Perl and in Unix-like operating systems, and how they can accomplish tricky text manipulating tasks on the fly.

9.2 Building a Pattern

9.2.1 Getting your Foot Wet

Now you will learn to build a pattern using the regular expressions offered by Perl. To search for a pattern match, simply construct the pattern and put it in between the two slashes of the `m//` operator. If you don't need the bells and whistles, for example, you just need to know if the characters "`able`" appear in any given string, the pattern is as simple as:

```
m/able/
```

Let's put this to a test. Now, to see if this pattern occurs in the string "`Capable`", we bind the twos together by using the binding operator =∼. Try this script:

```
1 if ("Capable" =~ m/able/) {
2     print "match!\n";
3 } else {    # This should NEVER happen
4     print "no match!\n";
5 }
```

There is not many things special here. Because the pattern "`able`" is in the string "`Capable`", the words "`match!`" will be displayed. I intentionally use the literal "`Capable`" in the example to

show that although the symbol looks like an assignment operator, it is not necessary for a valid lvalue on the left hand side of the binding operator (remember lvalue is for assignment operators only). You may put any piece of scalar data, including scalar variable, in place of the string literal.

9.2.2 Introduction to m// and the Binding Operator

As you have seen in the above example, the m// operator is used for pattern matching. In between the forward slashes // the pattern to match is placed. Additional options, if any, are placed at the end after the last slash. If an expression is explicitly bound to the operator using the =~ or !~ binding operators, that expression is searched for the pattern specified. If the binding operator is missing, as you will see in some later examples, =~ is assumed and $_ is taken as the expression to be searched.

In scalar context, the binding operator =~ returns a true value if the expression matches the pattern, an empty string (and hence a false value) if otherwise. !~ simply inverts the logic so that if the expression matches the pattern a false value is returned, a true value otherwise. Therefore, the following two expressions are equivalent:

```
!($expression =~ m/pattern/)
$expression !~ m/pattern/
```

Similar to double-quoted strings, the pattern may be interpolated. Therefore, you can generate patterns at runtime and apply them by, for example,

```
$expression =~ m/$var/
```

Again, similar to the case of quoted strings, you may use other symbols in place of //. However, if you use //, you may omit the prefix m. You may wish to use other symbols in place of // if your pattern is heavily slashed, for example, to match a Unix path name /var/logs/httpd/error_log in an expression you have to escape the forward slashes (to be covered later) like this:

```
$expression =~ m/\/var\/logs\/httpd\/error_log/
```

In the manual pages, this is described as the *leaning toothpick syndrome* (LTS) where a lot of forward and backward slashes are present, making the pattern itself difficult to recognize. If you change the symbol to, for example, |, then the entire pattern suddenly becomes clear:

```
$expression =~ m|/var/logs/httpd/error_log|
```

This is just one of the methods to remove the leaning toothpick syndrome. We will talk about the m// operator in more detail later in this chapter, together with the options you may use. I am just giving you a taste of it now anyway.

Metacharacter	Default Behaviour
\	Quote next character
^	Match beginning-of-string
.	Match any character except newline
$	Match end-of-string
|	Alternation
( )	Grouping and save subpattern
[ ]	Character class

Table 9.1: Metacharacters in Perl

9.2.3 Metacharacters

The list of metacharacters supported in Perl are listed in Table 9.1.

Metacharacters serve specific purposes in a pattern. If any of these metacharacters are to be embedded in the pattern literally, you should quote them by prefixing it by \, similar to the idea of escaping in double-quoted string. In fact, the pattern in between the forward slashes are treated as a double-quoted string. For example, to match a pair of empty parentheses and execute a code block if they can be found, the code should look like

```
if ($string =~ m/\(\)/) {
    # ...
}
```

In the previous section we mentioned the leaning toothpick syndrome. Apart from changing the delimiters of the `m//` operator, you can suppress the effect of metacharacters by using the `\Q ... \E` escape sequence. This does not suppress interpolation, however. This is demonstrated in the following example:

```
$expression =~ m/\Q/var/logs/httpd/error_log\E/
```

| specifies alternate patterns where matching of either one of them results in a match. These patterns are tried from left to right. The first one that matches is the one taken. Usually, | are used together with parentheses ( ) to indicate the groupings preferred. These are some example patterns:

```
m/for|if|while/     # A match if either 'for', 'if' or 'while' found
m/a(a|b|c)a/    # A match if either 'aaa', 'aba' or 'aca' found
```

Apart from indication of grouping, the use of parentheses also carries another behaviour. If there is a pattern match, the expression matched by a grouped pattern is saved. This is called **backtracking**. Backtracking is covered in more detail later in this chapter.

The . metacharacter matches any character. By default, it does not match any embedded newline characters in a multi-line string. However, if the `s` option of `m//` is given, embedded newline characters will be matched. This is convenient if you have to match a pattern across multiple lines.

```
"a\nb\nc" =~ m/a.b/     # Not matched, because . does not match \n
"a\nb\nc" =~ m/a.b/s    # Matched with 's' option
```

The ^ metacharacter matches the beginning of the string, and $ matches the end of the string. However, if the `m` option of `m//` is given, they match the beginning and the end of each line respectively. This is used to match individual lines inside a multi-lined string.

```
"a\nb\nc" =~ m/^a$/     # Not matched
"a\nb\nc" =~ m/^a$/m    # Matched
```

9.2.4 Quantifiers

Quantifiers are used to specify how many times a certain pattern can be matched consecutively. A quantifier can be specified by putting the range expression inside a pair of curly brackets. The format of which is

```
{m[,[n]]}
```

Here are the available variations:

```
{m}     Match exactly m times
{m,}    Match m or more times
{m,n}   Match at least m times but not more than n times
```

This example shows how you can verify if a string is an even number. Note that this example can be further simplified with the help of character classes, which we will describe next.

```
$string = $ARGV[0];
my $retval = ($string =~ m/^(\+|-){0,1}(0|1|2|3|4|5|6|7|8|9){0,}(0|2|4|6|8)$/);
printf("$string is%san even integer.\n", $retval?' ':' not ');
```

With different input values, different messages will be printed indicating whether the number is an even integer. You may split the pattern into three sections. The first part, `(\+|-){0,1}` matches the preceding sign symbol if there is one. Note that the minimum number of times is 0. Therefore, this part still matches if the sign symbol is absent. Right after the optional sign symbol are the digits. We establish that an even number has the least significant digit being 0, 2, 4, 6 or 8. Therefore, on the far right we specify it as the last digit. In between the sign symbols and the least digit there can be zero or more digits. This is how we arrive at this pattern.

Perl defines three special symbols to represent three most commonly used quantifiers. * represents `{0,}`; + represents `{1,}` and ? represents `{0,1}`. Because + is a quantifier as a result, it has to be escaped in the example pattern above.

9.2.5 Character Classes

A character class includes a list of characters where matching of any of these characters result in a match of the character class. It is similar in some sense to alternation, but the way they are interpreted is different. A character class is constructed by placing the characters inside a pair

of square brackets. Here I demonstrate how to rewrite the pattern in the above example using character classes.

```
my $retval = ($string =~ m/^[+-]?[0123456789]*[02468]$/);
```

It's a lot shortened. Isn't it? All characters that appear inside the square brackets belong to one character class. We have also used the special quantifier symbols described above to further shorten the pattern. But that's not the end. You can further shorten the character class comprising all digits by specifying in the form of a range:

```
my $retval = ($string =~ m/^[+-]?[0-9]*[02468]$/);
```

You may define multiple ranges in a character class, for example, `[a-zA-Z]` matches all lowercase and uppercase forms of English alphabets.

Inside a character class, if you prefix the list of characters with `^`, that means any characters that are not listed results in a match. For example, `[^0-9]` matches any character provided it is not numeric.

Perl also defines some special character classes that contain lists of common character combinations in pattern matching.

Character Class	Content
`\w`	Alphanumeric characters and _ (`[a-zA-Z0-9_]`)
`\W`	Neither alphanumeric characters nor _ (`[^a-zA-Z0-9_]`)
`\s`	Whitespace characters (`[ \t\n\r\f]`)
`\S`	Non whitespace characters (`[^ \t\n\r\f]`)
`\d`	Numeric digits (`[0-9]`)
`\D`	Non numeric digits (`[^0-9]`)

Table 9.2: Special Character Classes in Perl

Finally, our example pattern to match even integers can be simplified as

```
my $retval = ($string =~ m/^[+-]?\d*[02468]$/);
```

which is now the most compact form you can attain.

9.2.6 Backtracking

Parenthesised patterns have a useful property. When pattern matching is successful, the matching substrings corresponding to the parenthesised parts are saved, which allow you to save them for further operations. For example,

```
$string = 'Telephone:   1234-5678';
if ($string =~ m/^Telephone:\s*(\d{4}-\d{4})$/) {
    print "The telephone number extracted is '$1'.\n";
}
```

In this example, the telephone number extracted is saved as $1. There can be multiple bracketed patterns in a given pattern. The matched substrings are numbered in ascending order of position of the opening parentheses. If we change the pattern as follows:

```
$string = 'Telephone:   (852) 1234-5678';
if ($string =~ m/^Telephone:\s*(\((\d+)\)\s*(\d{4}-\d{4}))$/) {
    print "The telephone number extracted is '$1'.\n";
    print "The country code extracted is '$2'.\n";
    print "The local phone number extracted is '$3'.\n";
}
```

```
The telephone number extracted is '(852) 1234-5678'.
The country code extracted is '852'.
The local phone number extracted is '1234-5678'.
```

The pattern looks more complicated then it really is. If you examine it carefully, there are three bracketed patterns in it. The first one embracing the telephone number in full, including the country code. The second and third are placed inside this bracket to extract the country code and local phone number separately. The positions of the opening braces determine the ordering. Therefore, we can observe that in case of nested parentheses, the outer one has a smaller number than the inner one.

There is a very important property with respect to quantifiers that you have to be aware of. Quantifiers default to being "greedy". That is, quantifiers try to match as many times as possible. For example when you match the string "greediness 2003" against the pattern `/(.*)(\d+)/`, $1 has the string "greediness 200", leaving "3" to $2. Therefore, you can see that quantifiers appearing on the left tries to match as many times as possible while allowing the pattern to match. To work around this situation, you may append a `?` character to the quantifier to suppress this greedy behaviour. Therefore, for the sample string the best pattern is `/(.*?)(\d+)/`, which separates all numeric digits at the end from the rest of the string.

9.3 Regular Expression Operators

9.3.1 `m//` — Pattern Matching

As we have been using so far, the `m//` operator performs pattern matching. It supports a number of options. We have covered `m` and `s`, and now it's time for a revision. The `s` option treats the string being searched as if it consists of a single line only. By doing so, `.` will match an intermediate newline character. The `m` option allows matching of individual lines in a multi-line string. Here, I will introduce several other commonly used options.

The `i` option matches in a case-insensitive manner. Therefore,

```
'ABCD' =~ m/abc/i
```

results in a match. By default, pattern matching is case sensitive. Another useful option is `g`, which attempts to carry out a global pattern matching on the string. In scalar context, a search pointer is maintained. The search pointer is first initialized to the beginning of the string. In each matching operation, matching starts from the search pointer. If a match is found, the search pointer advances

to past the end of the matched substring. If matching fails, the search pointer is reset to the initial position. You can use the `pos()` function to retrieve the position of the current search pointer.

You can use this option to find out the position of occurrences of certain patterns in the string. The following example illustrates this point:

```
$string = 'Telephone: 1234-5678';
while ($string =~ m/(\d{4})/g) {
    print "'$1' found at position " . (pos($string) - length($1)) . ".\n";
}
```

```
'1234' found at position 11.
'5678' found at position 16.
```

In list context, the `m//g` operator (with `g`) returns a list consisting of all parenthesised substrings from the matching. Therefore,

```
my $string = 'Telephone: (852) 1234-5678';
my @list = ($string =~ m/^Telephone:\s*(\((\d+)\)\s*(\d{4}-\d{4}))$/g);
# @list = ('(852) 1234-5678', '852', '1234-5678')
```

results in the list

```
@list = ('(852) 1234-5678', '852', '1234-5678')
```

9.3.2 s/// — Search and Replace

This operator is a powerful search-and-replace engine that you can use to flexibly search for certain patterns and replace it with a replacement string. The first argument is the search pattern, just as the case of `m//`. The second argument is the replacement string. As you will soon see, backtracking is immensely useful in this regard.

The options that I mentioned above that applies to `m//` also apply to `s///`. But there is a new one. The option `e` causes the replacement string to be treated as an expression instead of a double-quoted string. That is, you can use a combination of operators to generate the desired replacement string at runtime.

Without the option `g`, only the first occurrence of the pattern is replaced. With the option `g`, all occurrences of the pattern are replaced in one go.

Here are some examples:

```
$string =~ s/\t/' ' x 4/eg;     # change all tabs to 4 spaces
$string =~ s/^(.*)\n$/$1/s;     # like chomp(), to remove trailing newline
```

9.3.3 `tr///` — Global Character Transliteration

`tr///` is a convenient and efficient operator that changes a set of characters into another. The first argument is the character list to search for. The second argument is the character replacement list. It builds a character translation map at compile time. At runtime, it changes any characters that can be found in the string into the corresponding character in the replacement list. For example,

```
tr/a-z/A-Z/
```

is just an alternative way to convert characters to uppercase form without using `uc()`.

9.4 Constructing Complex Regular Expressions

While the general principles of regular expressions are not complicated, constructing the correct pattern for your specific purpose is not itself a simple affair, especially if it is a complex one. An incorrect pattern is one which does not match a string intended to match, or match a string that should not be matched. Despite experience develops over time, following a formal, systematic procedure helps you construct regular expressions with a higher degree of accuracy.

The mechanism is known in the academic community as **finite-state automaton** (FSA). The definition is pretty abstract but the idea is rather easy to understand with the help of examples. Basically, we construct an FSA diagram. The diagram has a number of states, with each of the states linked together by arrows which represent transitions. This diagram is also known as a **state diagram**. At any point in time an FSA system can only be in one of the states. Every input triggers a transition, which may or may not change the state of the FSA.

As an example, we will construct a pattern which matches all relative path expressions. For example, `images/fsa.gif` and `docs/perltut` are considered valid paths. However, we add in an additional requirement to invalidate paths containing "." and ".." as a path segment, because we will reuse the pattern constructed in the next chapter. We also assume each path segment consists of word characters (`\w`) only.

The constructed FSA diagram is shown below:

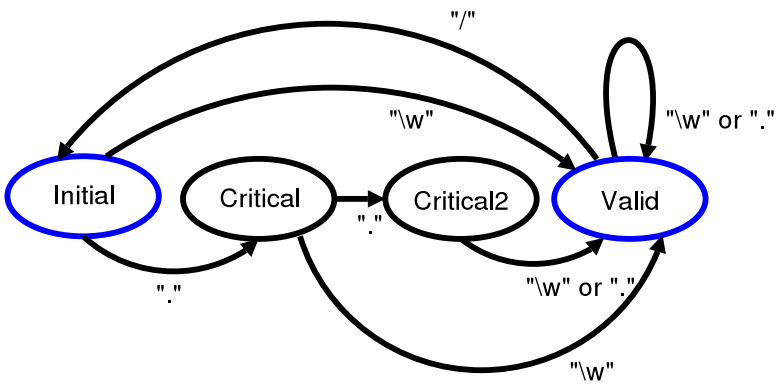

Figure 9.1: FSA to Match Valid Relative Paths

How do we interpret this diagram? We regard a path like `images/fsa.gif` as a sequence of characters. We break this path into its constituent characters, and feed each of them in sequence to the FSA. In this diagram, each of the states is represented by an oval with the name of the state written at the centre. The initial state is marked as "Initial", which is the state before any characters are fed to the system. An arrow represents transition and beside each arrow the character which triggers the transaction is indicated. We also define that the states marked in blue are valid terminate states, namely "Initial" and "Valid". At the end of the input stream, if the system is at a valid terminate state, then the string matches our criteria. Now "i" is fed to the system. Because "i" is a member of `\w`, a transition to the "Valid" state occurred. Therefore, the state is now "Valid". Then, the character "m" triggers a loopback to the "Valid" state. Similar are the characters "a", "g", "e" and "s" that follow. The next character, "/" triggers a transition to the "Initial" state. You may follow the remaining characters in a likewise manner and eventually, after the last character "f" you will be at the state "Valid". This is the terminate state, and because it is a valid terminate state, this path matches our criteria.

Note that any input character not matching any transitions from the current state shown in the diagram automatically triggers a transition to a hidden illegal state, which is not shown in the diagram. This is because we only emphasize valid transitions, not invalid ones.

This diagram was constructed based on the idea that we may break a path into its constituent path segments. We start at the initial state. While constructing this diagram, we have to identify that we need to have special treatment on path segments that start with a dot. We can accept path segments that start with `\w`. Therefore, we can immediately transit to the "Valid" state from the initial state if it is a word character. If it is a dot, then we follow another path to the "Critical" state. If the next character is also a dot, then we transit to the "Critical2" state. Word characters appearing afterwards bring the system back to the "Valid" state. From the diagram, it can be seen that we allow path segments such as "..." and "..somedir", but not "..". Depending on the operating system, you may wish to change this assumption if this is not acceptable to you (e.g. on Windows three consecutive dots represent two levels up the directory tree). Note that at the end of each path segment, a "/" brings the system back to the "Initial" state. This allows paths consisting of an arbitrary number of path segments. In order to be more flexible, the path may or may not be terminated with "/". Therefore, the "Initial" state has to be made a valid terminate state instead of just the "Valid" state.

With a correct FSA established, rewriting it into a pattern becomes very easy. We only have to consider paths from the initial state that lead to the valid states. Note that in this example we may follow three paths to reach the "Valid" state. The first path is with the input of a word character, that is `\w`. The second path passes through the "Critical" state only, consisting of a dot followed by `\w`, and hence the path is `.\w`. The third path passes both critical states, that is `..[.\w]`. Therefore, expressed in regular expression format these three paths can be represented by `\w|\.\w|\.\.[.\w]`. At the "Valid" state, there is a transition back to itself. That means a character of the character class `[.\w]` does not trigger any state changes. This implies you can inject as many characters of this character class without changing the state of the FSA. In regular expressions, this is synonymous to `[.\w]*`. Finally, please note that the "Initial" state is also a valid terminate state. Therefore, "/" which transits to the "Initial" state is optional. With all these combined, the regular expression representing a valid path segment is

```
/(\w|\.\w|\.\.[.\w])[.\w]*\/?/
```

Because this represents just one path segment, finally we arrive at the following pattern to match full relative paths:

```
/^((\w|\.\w|\.\.[.\w])[.\w]*\/?)+$/
```

Construction of an FSA forces you to think about the specifics of the parts that make up the pattern carefully. For more complicated patterns, this method generally yields more accurate patterns than pure imagination in your brain because it is easier for you to verify whether the pattern is correct with the FSA. Usually you may not get the FSA correct at the very first time, but through corrections and refinements you can come up with a very accurate pattern. When you are highly experienced with regular expressions, it is likely that you do not have to follow this elaborate procedure anymore.

Summary

- Pattern matching refers to the operation of comparing a string against a given pattern to determine if the string matches the pattern.
- In Perl, a pattern is specified in the form of a regular expression. Regular expression is supported in many programming languages and programs.
- To perform pattern matching in Perl, a string is bound to the pattern with the binding operator =~ or !~.
- Metacharacters serve special purposes in a pattern.
  - \ quotes the next metacharacter so that it is interpreted literally.
  - ^ refers to the beginning of a string.
  - $ refers to the end of a string.
  - . refers to any character except newline character.
  - | lets you specify alternative subpatterns. Matching is successful if any of the subpatterns match.
  - ( ) are used to group subpatterns. Matched subpatterns are also saved in special variables starting from $1 for backtracking.
  - [] establishes a character class. It includes a list of characters where matching of any of these characters result in a match of the character class. If prefixed by ^, the character class matches any character not appearing inside the square brackets.
    - \w represents a character class containing alphanumeric characters and _
    - \W represents [^\w]
    - \s represents a whitespace character
    - \S represents any non-whitespace character, i.e. [^\s]
    - \d represents a numeric digit
    - \D represents any non-numeric character
  - {} is a quantifier which specifies how many times a certain pattern can be matched consecutively.
    - * implies {0,}, i.e. zero or more times
    - ? implies {0,1}, i.e. none or single time
    - + implies {1,}, i.e. one or more times
    - A ? character following the above quantifiers disables greedy mode.
- The m// operator performs pattern matching.
  - The m option allows m// to match individual lines in a multi-line string. ^ and $ match the beginning-of-line and end-of-line respectively.
  - The s option causes . to match any newline characters.

  - the `i` option performs case-insensitive pattern matching.
  - The `g` option enables global pattern matching on the string.

- The `s///` operator performs search and replace, which supports the options `m`, `s`, `i` and `g` with the same semantics as `m//`, the option `e` is supported which causes the replacement string to be treated as an expression.
- The `tr///` operator performs global character transliteration on a string.

Questions

A. Modify the FSA diagram to disallow path segments matching the pattern `/^\.+$/` and construct the corresponding match pattern.

B. In Chapter 8 we developed a class `FileSearch` to search for a file recursively in a directory tree whose name matching a pattern specified by the user. Now, use this class to write an application which finds out all classes accessible by Perl and their respective locations on the filesystem. For example, selected lines of output on my system look like this:

```
Bundle::Apache2 (/usr/local/lib/perl/5.8.0)
Bundle::CPAN (/usr/share/perl/5.8.0)
Bundle::DBD::mysql (/usr/local/lib/perl/5.8.0)
Bundle::DBI (/usr/local/lib/perl/5.8.0)
Bundle::LWP (/usr/share/perl5)
ByteLoader (/usr/lib/perl/5.8.0)
CGI (/usr/share/perl/5.8.0)
CGI::Apache (/usr/share/perl/5.8.0)
CGI::Carp (/usr/share/perl/5.8.0)
CGI::Cookie (/usr/share/perl/5.8.0)
CGI::Pretty (/usr/share/perl/5.8.0)
CGI::Push (/usr/share/perl/5.8.0)
CGI::Switch (/usr/share/perl/5.8.0)
CPAN (/usr/share/perl/5.8.0)
```

Note that several versions of the same class may appear on the filesystem. The path has to reflect the location of the version that will be loaded by Perl.

Chapter 10

Runtime Evaluation & Error Trapping

10.1 Warnings and Exceptions

Man is fallible. So are the programs written. In the course of compilation and execution of a program various kinds of diagnostic messages are generated. You have for sure encountered a lot of these in your learning process. This chapter deals mainly with errors and error handling. Before proceeding to the rest of the text, it is discreet for us to establish a common terminology first.

We may classify an error by the time at which it occurs. They can be either **compilation errors** or **runtime errors**. When a Perl source file is being compiled, **syntax errors** are being discovered as part of the parsing process. Every language follows a certain set of syntax, which governs how different parts of the language can be put together. Syntax errors are thus compilation errors. When a compilation error occurs, the compilation process will be terminated. Errors are also generated when a source file is being executed. We collectively refer to them as runtime errors. For example, division by zero is a runtime error.

We may also classify an error by its severity. Some errors are severe enough that the program should not be allowed to continue. In Perl manpages, this is customarily called **fatal exceptions**. For example, division by zero is a fatal exception. There are also some kinds of diagnostic messages that are generated, but will not terminate the program. They are called **warnings**. For example, trying to `print()` the value of an undefined variable if the `-w` switch is in effect results in a warning. Generation of warnings may not be fatal for your program, but they usually signify there may be some subtle logical errors in your program. You should try to find out why they occur and try to eliminate them, or at least you have to ensure that these warnings are expected.

Note that all errors and warnings are customarily redirected to the standard error, which is the screen by default unless you redirect it to somewhere else, as indicated in Chapter 8. For CGI scripts, these messages are usually recorded in Web server log files that you can examine to help locate sources of errors.

10.2 Error-Related Functions

Perl provides you with two builtin functions to generate fatal exceptions and warnings, namely `die()` and `warn()` respectively. Both of them accept one parameter which is usually the message associated with the exception or warning. If multiple arguments are given, they are simply concatenated to form a single error message. If the error message is not terminated with the newline

character, the file name and line number are appended. This is to make debugging easier.

Usually you use `die()` to signal occurrences of fatal errors in the sense that the program in question should not be allowed to continue. For example, I mentioned in Chapter 8 that you should check the return value of `open()` to see if file open is successful before proceeding to other file operations. Perl programmers customarily do this by a statement like

```
open FILE, "<file.txt" or die "Cannot open file: $!\n";
```

It uses the short-circuiting behaviour of the `or` operator. Only if `open()` fails and returns an `undef` value that `die()` is called. `$!` is a predefined variable that returns the error message returned by the system on the error. Because failure of `open()` does not constitute a fatal error, for some programs it may be deemed necessary or appropriate to manually terminate it (however, most sophisticated programs prefer to handle the error and skip the relevant parts of the program instead of abruptly halting it). The `die()` function serves this purpose.

The `warn()` function generates a warning instead of a fatal exception. It merely outputs the message to the standard error and does not terminate the program.

10.3 `eval`

Consider one day you are asked to write a command-line application that allows users to input a mathematical expression and the program will then calculate the result and display it to the user. This seems like an easy task, but it isn't. You may still manage it if you impose restrictions to allow only simple expressions like `3 + 4` and `4 * 6`. But what if the expression is immensely complex? The program has to parse the expression. The expression is also checked for syntax errors and evaluated according to the operator precedence. Parsing a complicated expression is not an easy task without specialized knowledge in compiler construction. It turns out that you may construct a Perl statement at runtime and evaluate it on-the-fly. Therefore, what you need to do is to have the user specify the expression according to Perl syntax, and you simply invoke the function `eval()` to evaluate it. This is very convenient because Perl checks the syntax and evaluate it for you, so you don't have to.

The program can thus be as simple as this:

```
1  # WARNING: GRAVE SECURITY HOLE!!!
2  print "Please enter the expression below:\n";
3  chomp(my $expr = <STDIN>);
4  my $result = eval($expr);
5  print "The return value is $result.\n";
```

`eval()` accepts a string as its argument which is parsed as a Perl program. In the above example, if the user enters

```
3*(4+5)
```

then this is evaluated, so the evaluated result, 27, is returned and assigned to `$result`. Note that you may put multiple statements in the argument to `eval()`, delimited by `;`, to evaluate multiple

statements in one go. The final ; may be omitted, as illustrated in the example.

In this form, both parsing and evaluation occur at runtime, because the code to be evaluated are not available until at runtime. eval has another form. Instead of passing to it a string containing the code, a code block is passed embracing the code (note the final semicolon after the code block!). This form is different from the above form in that because the code itself is static, it can be parsed at compile time. Therefore, you cannot dynamically construct a Perl program and evaluate it with this form. For example,

```
eval {
    $result = $a / $b;
};
```

Then you may wonder what is the use of the second form if the statement(s) inside an eval{} block are parsed statically just as if the eval{} is not in effect. In fact, both forms of eval may be used for error trapping. In the first form, syntax errors as well as runtime errors occurred when evaluating the code will be trapped. The error message is sent to the predefined variable $@, and eval() returns with the value undef. In the second form, compile-time errors cannot be trapped so they are still displayed and compilation is halted. Runtime errors will be trapped and redirected to $@ similar to the first form.

Although the first form of eval() is very convenient because it can evaluate a string as if it is a tiny Perl program, it is also highly dangerous if it is cleverly abused. You will see why shortly after we cover backticks, an equally powerful and dangerous tool.

10.4 Backticks and system()

You may execute operating system commands in Perl as if you are on the command line of your operating system. The system() function accepts a list of strings as parameters. The first argument is the name of program, followed by the arguments. For example,

```
system('notepad.exe', 'myfile.pl');
```

On a Windows system, this executes the Windows Notepad and opens the file myfile.pl. The system() function doesn't return until the program terminates.

Backticks ` ` are more flexible as it simulates the command line. What you need to do is to construct the command and enclose it in backticks instead of single or double quotes. In US keyboards, the backtick is located underneath the Escape key, sharing the key with the tilde (~). An example is

```
$output = `ls -l`;      # get long directory listing
```

Text that are sent to the standard output are collected and assigned to $output.

10.5 Why Runtime Evaluation Should Be Restricted

While `eval()` is an immensely powerful tool for programmers, it is also one of the most dangerous weapon frequently taken advantage by abusers to crack the system. Sometimes it may even be possible to launch catastrophic attacks. When combined with backticks, this is even more dangerous. Consider this example, which we used to evaluate mathematical expressions on-the-fly:

```
1 # WARNING: GRAVE SECURITY HOLE!!!
2 print "Please enter the expression below:\n";
3 chomp(my $expr = <STDIN>);
4 my $result = eval($expr);
5 print "The return value is $result.\n";
```

What an attacker has to enter is `‘rm -fR /‘`, which is the Unix command for deleting everything in the root directory. If this script is executed by root, the hard disk will be wiped spick-and-span in a few minutes; Otherwise, at least the home directory of the user running the program will be deleted. On Windows, passing a `rmdir` command is equally devastating. Therefore, as you can see, `eval()` poses a significant security threat if improperly used.

10.6 Next Generation Exception Handling

10.6.1 Basic Ideas

While `eval{}` is the traditional method of exception handling in Perl, it suffers some grave drawbacks. The major drawback is that with a deep call stack and the exception is caught at a deep level, you usually have to put the error message as the return value and `return()` it multiple times as you exit from each level in order to notify the error to a caller at the outermost level. Having to always manually check the return value of `eval{}` is not convenient at all.

If you have used certain languages like Java and C++ you may know that exception handling in those languages are performed differently. The mechanism is called a try/catch. While the Perl language does not officially support this exception handling mechanism, some wise people managed to write a Perl module that satisfactorily implemented a similar mechanism from mere Perl code. The module is on the CPAN, with the name `Error`. The latest version as of this writing is 0.15. It is not in the Perl distribution so you have to install it as described in Appendix B. It is a pure Perl module, so you may easily distribute it with your Perl program if you use it in your program. It is also expected that this exception handling mechanism, possibly with slight adjustments, will be officially included in the Perl 6 language. This will not come in the near future, though. For your convenience, I have bundled the `Error.pm` module together with my code examples.

As of this writing, the documentation of the latest version of `Error.pm` can be found at http://search.cpan.org/author/UARUN/Error-0.15/Error.pm.

To use this module, simply `use()` the module and import the symbol `:try`, i.e.

```
use Error ':try';
```

I hereby start my discussion with the `try {}` block. Similar to `eval {}`, the code inside the curly braces are executed. In fact, internally, `try{}` uses `eval{}` to handle the exceptions. Therefore, it is

compatible with `eval{}`. Let us see how we may use `try {}` in place of `eval {}` for the following simple case:

```
my $retval = eval {
    open FILE, "<file.dat" or die "file.dat cannot be opened: $!";
    while (<FILE>) {
        # Do something
    }
    close FILE;
};
if (!defined $retval) {
    print "ERROR: $@\n";
}
```

We may transform it into the try/catch way by:

```
use Error ':try';

try {
    open FILE, "<file.dat" or die "file.dat cannot be opened: $!";
    while (<FILE>) {
        # Do something
    }
    close FILE;
} catch Error::Simple with {
    my $err = shift;
    print "ERROR: $err";
};
```

Here, the `try` block is evaluated, and if the file `file.dat` cannot be found, the `die()` function raises a fatal exception. `try` is actually a subroutine defined in the `Error` package. When the exception is raised, an error object of class `Error::Simple` will be automatically created to represent the error. This is known as **throwing** the error.

After an exception is thrown, blocks after the `try` block are checked sequentially to search for a **handler** to **catch** the exception. After the `try` block you may place some handlers to handle any exceptions raised. A `catch` block is a handler which handles only a certain class of error. In the example above, the only `catch` block handles only error objects whose class is or inherits `Error::Simple`. Because the error object generated by `die()` in the `try` block belongs to this class, this handler is executed. The error message is `print`ed as a result. ***Note that you have to place a semicolon after the block of the last handler.*** This semicolon marks the end of the entire try/catch structure. Execution resumes afterwards.

What if a matching handler cannot be found in the current environment? The exception is passed immediately to the parent environment, if any. The current environment terminates. If the parent environment has a suitable handler, that handler is executed. Otherwise, the exception is passed to its parent environment etc. In other words, the exception travels up the call stack until it finds an appropriate handler, executes it and resumes execution right after the `try` block. Note that once a handler is complete, other handlers that follow will be skipped. Here is an example CGI script which converts the file "file2html.pl" (which is the script itself) to HTML for viewing in a browser. It uses

try/catch exception handling. You may execute this program from the command line. If you would like to view it in a browser, please consult the next chapter.

EXAMPLE 10.1 Text File to HTML Converter

```
#!/usr/bin/perl -w

use Error ':try';

# Forward declarations
sub openFile;
sub toHTML;

print "Content-Type: text/html\n\n";
print toHTML('file2html.pl');

sub openFile {
    my $mode = shift;
    my $filename = shift;

    local *FILE;
    open FILE, "$mode$filename" or die "File \"$filename\" cannot be opened: $¶
        !";
    return *FILE;
}

# convert to HTML
sub toHTML {
    my $filename = shift;
    my $html = "<html>\n<head><title>$filename</title></head>\n<body>\n";

    try {
        local *FILE;
        *FILE = openFile('<', $filename);
        $html .= "<pre>\n";
        my $line = '';
        while ($line = <FILE>) {
            $line =~ s/&/&amp;/g;
            $line =~ s/</&lt;/g;
            $line =~ s/>/&gt;/g;
            $line =~ s/\t/' ' x 4/ge;
            $html .= $line;
        }
        close FILE;
        $html .= "</pre>\n";
    } catch Error::Simple with {
        my $err = shift;
        $html .= "<h1>Error</h1><br>\n<div style='color: #FF0000'><b>$err</b></¶
            div>\n";
    };
```

```
44
45      $html .= "</body>\n</html>\n";
46  }
```

The script reads the content of itself by default. If you change the filename on line 10 to a non-existing file, then an error page will be generated instead.

In the default case, the `open()` should be successful, and hence the exception is not thrown. In this case, all handlers are skipped. If the file cannot be opened, the exception is thrown on line 17. Because the `openFile()` subroutine does not have any handlers defined, the subroutine terminates and the exception is passed to the parent environment. Note that `openFile()` is invoked inside a `try` block in the subroutine `toHTML()` (line 28). On receipt of an exception from `openFile()`, the rest of the `try` block is bypassed and the handlers are searched. Here, we have a matching `catch` handler. Therefore, it is executed. Note that handlers internally are also implemented by subroutines, and the error object is always passed as the first argument of a handler. In the handler, the error message is embedded into the HTML output. Once the handler completes successfully, execution resumes on line 45.

This example demonstrates the elegance of the try/catch approach. You do not have to handle an exception until you would like to. Traditionally, to notify the caller of a subroutine that an error has occurred inside the subroutine, the error string or error number has to be passed as the return value. Multiplexing normal return values and error numbers merely disturbs the program logic, and this is not convenient. It is even more problematic if you have to propagate the error number of string up the call stack. The try/catch mechanism generates and handles errors through another path so you do not have to mess with return values in order to perform exception handling.

If an error object is thrown and no matching handler can be found all the way up the call stack, the program terminates with the exception message displayed, similar to the behaviour of executing `die()` that is not inside any `eval`. In most well-written programs, this should generally not occur because you will usually have a generic handler that is executed as the last resort. More on this later on.

10.6.2 Throwing Different Kinds of Errors

The power of try/catch does not stop here. The ability to generate different kinds of exception objects is what that makes this mechanism immensely flexible. You won't need multiple handlers if you simply wish to handle `Error::Simple` exception objects. Only one handler as illustrated in the above example is already sufficient for this purpose.

All exception objects has to belong to a class that is a subclass of `Error`. In fact, `Error::Simple` is just a class that is predefined to represent simple errors. You can derive your own subclass to add extra functionality or record extra pieces of debugging information. You do not have to create an `Error` subclass for each single type of error. The general rule is, if you are likely to handle some kinds of errors differently than the others (by using different handlers), then you should create an exception class for each.

Here is an example. Because usually users of CGI applications are not programmers, you may wish to provide them with additional information other than the regular error messages. For example, in your error message you may include a suggested method to rectify the problem, tailored to the situation. I will now create an exception class that records this piece of extra information.

EXAMPLE 10.2 Subclassing Error.pm

```
1  # Exception.pm --- Sample exception class
2  # Reference:
3  # http://www.perl.com/pub/a/2002/11/14/exception.html
4
5  package Exception;
6
7  # The following is equivalent to
8  # use Error; @ISA = ('Error');
9  use base "Error";
10
11 # Operator overloading --- so that simple error message
12 # is displayed on string interpolation of error object
13 use overload ('""' => 'stringify');
14
15 sub new {
16     my $this = shift;
17     my $msg = shift;
18     my $soln = shift;
19
20     local $Error::Debug = 1;    # enable stacktrace
21
22     # Ensure correct discovery of position where error object thrown
23     # Needed because overriding of constructor
24     local $Error::Depth = $Error::Depth + 1;
25
26     $this->SUPER::new(-text => $msg, -soln => $soln);
27 }
28
29 # Show solution, if any
30 sub solution {
31     my $this = shift;
32     return $this->{'-soln'};
33 }
34
35 1;
```

Line 9 is an alternative way to say that this class is a subclass of `Error`. You can use the plain old `@ISA` method. I just teach you another way to do it. Line 13 actually utilizes **operator overloading**, a method of introducing new meanings to existing operators if the operands are objects. Here, this statement actually means if you place the object variable inside a double-quoted string, the object variable will be replaced by the return value of the method `stringify()`, which is also defined in `Error` returning a simple error message. This statement is added because we would not like to break this behaviour. Operator overloading is not covered in this edition of the tutorial. Line 20 enables the production of **stacktrace**. A stacktrace is a textual representation of the current call stack. From the stacktrace you will find at which point in the program an exception is raised, and the hierarchy of callers to the point in question. Programmers will find this information useful when debugging a program, but it may not be useful in production environments at all. Because generation of backtrace is not very efficient, this is disabled by default. in this class, we enabled it. Line 24 is placed to ensure that the point at which the exception is thrown is resolved correctly. I am

not explaining it here because it relates to some internal mechanisms that are not easy to explain with a few words. As a rule of thumb, just include it and you are fine.

The `Error` class is a hash reference. Fields that are carried with the object are stored in the hash itself. On line 26 we assign to it two fields. The `-text` field carries the basic error message, and `-soln` is the field added by us to hold the suggested solution. We also define a `solution()` method to get this piece of information from outside of the class. At last, we invoke the constructor of the superclass.

You may also wish to read the source code of `Error.pm`. It is not very difficult to read, and it is pretty short. You may find out more information about the workings of the module that are absent from its documentation. In particular, you may read the source code of `Error::Simple`, which is in the same file as `Error.pm`. It gives you some hints as to how to write a subclass of `Error`.

Now I include a modified version of the text-to-HTML converter program that uses the `Exception` class we developed above. Apart from the error message, it displays the stacktrace as well as the suggested action in response to the error. A screenshot showing the generated error page is in Figure 10.1.

```
1  #!/usr/bin/perl -w
2
3  use Exception ':try';
4
5  # Forward declarations
6  sub openFile;
7  sub toHTML;
8
9  # Define a constant
10 use constant FILEIO_ERR => 'Please check if the specified file exists.';
11
12 print "Content-Type: text/html\n\n";
13 print toHTML('file2html.pl');
14
15 sub openFile {
16     my $mode = shift;
17     my $filename = shift;
18
19     local *FILE;
20     open FILE, "$mode$filename" or
21         throw Exception("File \"$filename\" cannot be opened: $!", FILEIO_ERR);
22     return *FILE;
23 }
24
25 # convert to HTML
26 sub toHTML {
27     my $filename = shift;
28     my $html = "<html>\n<head><title>$filename</title></head>\n<body>\n";
29
30     try {
31         local *FILE;
```

```
32          *FILE = openFile('<', $filename);
33          $html .= "<pre>\n";
34          my $line = '';
35          while ($line = <FILE>) {
36              $line =~ s/&/&amp;/g;
37              $line =~ s/</&lt;/g;
38              $line =~ s/>/&gt;/g;
39              $line =~ s/\t/' ' x 4/ge;
40              $html .= $line;
41          }
42          close FILE;
43          $html .= "</pre>\n";
44      } catch Exception with {
45          my $err = shift;
46          $html .= "<h1>Error</h1><br>\n<div style='color: #FF0000'><b>$err</b></¶
                div>\n";
47          $html .= "<h2>Suggested Action</h2><br>\n<div>" . $err->solution() . "</¶
                div>\n";
48          $html .= "<h2>Stacktrace</h2><br>\n<pre>\n" . $err->stacktrace() . "</¶
                pre>\n";
49      } catch Error::Simple with {
50          my $err = shift;
51          $html .= "<h1>Error</h1><br>\n<div style='color: #FF0000'><b>$err</b></¶
                div>\n";
52      };
53
54      $html .= "</body>\n</html>\n";
55  }
```

Here, you see an example of using multiple handlers, although only the handler of `Exception` is used in this example. Note that the handler of `Exception` can display more debugging information than the generic one. In the screenshot we can see a message indicating the course of action that should be taken, followed by the stacktrace.

Line 21 has been changed from `die()` to `throw()`. The `throw()` function is used to generate a new instance of the exception object and throw it. In this example, the package name of the exception object is `Exception` and the two parameters, the basic error message and the suggested action message, are provided, in accordance with the constructor of `Exception` we described earlier.

Line 10 declares a **named constant**. While Perl does not intrinsically has a mechanism for defining named constants, the `constant` module provides you with a nice workaround. Here, the constant `FILEIO_ERR` refers to the string on the far right. By using constants instead of scalar variables to store immutable data, you can avoid inadvertently modifying their contents because attempts to modify their values result in compile-time error.

10.6.3 Other Handlers

Now that you have learned the basic knowledge needed to make good use of exception handling. To end up this topic I will introduce to you the other kinds of handlers you can place to handle exceptions.

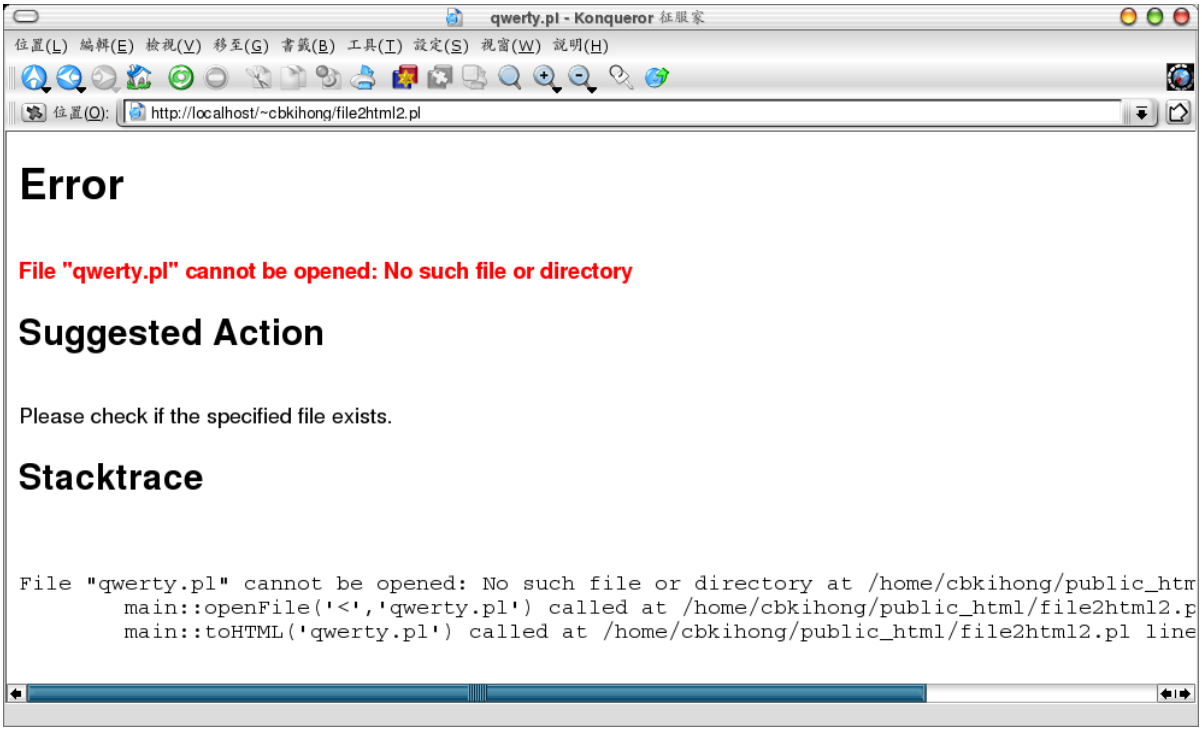

Figure 10.1: Web Page Showing Generated Exception

The `otherwise` handler handles all kinds of exception objects. It is generally used as a generic exception handler that acts as a last resort to catch the exception if an appropriate handler is missing. Therefore, you should ensure that this is placed after all `catch` and `except` handlers, if any. There can be at most one `otherwise` handler per `try` structure.

An `except` handler is rarely used in practice, because its functionality can largely be achieved using `catch` handlers. Please read the documentation if you are interested in knowing more about it.

A `finally` handler is different from the other handlers in that, if one is present, the block is evaluated if the `try` block is executed successfully (with no exceptions thrown) or if a handler has completed.

You may throw an exception object in a handler. For example, you may have a handler in the current environment to handle the exception. However, if the caller also has to be informed of this exception, you may simply `throw()` it again in the handler after you have carried out all necessary processing. For example,

```
...
} catch Error::Simple with {
    my $err = shift;
    # Do some processing here w.r.t. the exception
    $err->throw();      # let caller catch it also
};
```

10.7 Other Methods To Catch Programming Errors

Not all errors are fatal exceptions. More often logical or semantic errors arise because of mistaken (but syntactically acceptable) use of data values. For example, you may think a certain environment variable contains valid data, but for unknown reasons it turns out that it is actually `undef`. You are likely to get unexpected results if you keep on using this value without knowing it is `undef`. Certain kinds of errors like this may be trapped by using the following methods.

10.7.1 The `-w` Switch — Enable Warnings

You have been using this switch throughout this tutorial. By enabling this switch, warnings on certain dubious operations are generated. Here, I list some sample situations where warnings will be generated if warnings are enabled:

- ⋆ Using the value of a variable without first giving it a value (or the value `undef`);
- ⋆ Applying mathematical operators on an operand which is not a number (e.g. `"30ab" + 4`);
- ⋆ Variables that are mentioned only once. Usually, this signals a typographical error. The rationale is that in practice a variable is mentioned at least twice, that is when it is given a value (define) and when it is used. Any variables not adhering to this define-and-use pattern are rather likely misused;
- ⋆ The use of undefined filehandles. This usually indicates you have not check the return value of `open()` to see if the open is successful;

This is only a selection of the most frequent sources of warnings. Note that this switch determines whether warnings are generated by Perl internally. Therefore, warnings manually generated by `warn()` are not suppressed in the absence of this switch.

10.7.2 Banning Unsafe Constructs With `strict`

You can `use strict` to disable some unsafe constructs in Perl. It consists of three kinds of checks, whose imported symbols are represented by `refs`, `vars` and `subs`, respectively. By default, if you insert `use strict;` in your program, all of them are performed by default. You may selectively enable some of them but not the others by specifying the corresponding symbols after `use strict`. For example,

```
use strict 'refs', 'subs';      # Omitting 'vars'
```

If the `refs` check is in effect, it disables the use of so-called **symbolic references**. While it bears the name "references", it does not have much to do with references, or "hard references" as are called by some people described in Chapter 6. Symbolic references are generally considered undesirable and should be avoided. The idea is quite simple. Consider this example:

```
$b = 10;
$a = \$b;
print ${ $a }, "\n";
```

10 is printed. Recall that anything returning a reference may be placed inside the curly braces. This is the basis for dereferencing. However, a symbolic reference is just a plain string which contains the name of another *non-lexical* variable instead of a reference. Here is an example which uses symbolic reference that is largely functionally equivalent to the above example:

```
$b = 10;
$a = "b";
print ${ $a }, "\n";
```

The difference is on line 2, that is, the value held by the lexical variable $a. The first example is a proper reference, but the second one is not. If `refs` check is in effect, the second example will be flagged as invalid and a runtime error generated, because you are trying to dereference a string instead of a real reference. Note that symbolic references always resolve to non-lexical variables. For instance, in the above example $b is accessed from the symbol table. If you declare $b as lexical with `my` on line 1, Perl will still resolve it from the symbol table, which is `undef` unless you have it defined somewhere before. Because the use of non-lexical variables is discouraged, and you may potentially modify the value of the Perl predefined variables which hold important runtime data, symbolic reference should be avoided.

The `subs` check, despite its name, is used to trap the use of so-called **barewords**. As you may appreciate from your journey through this tutorial, Perl gives you a lot of shorthands that you may use to make your code more "compact", which implies a rather informal way of expression that lets you save some keystrokes provided you know exactly what you are doing.

A bareword is a standalone word in Perl that does not have any special meaning. For example,

```
my $word = something;
```

When parsing the assignment statement during compile time, Perl first checks if a subroutine of the name `something` has already been declared, because a subroutine may be invoked without having & and the parentheses. If it is not, then it is assumed to be a string with the quotes omitted. However, if

```
sub something {
    # something ....
}
my $word = something;
```

Then, `something` is treated as a subroutine invocation as a subroutine of this name exists when the assignment statement is parsed. If you put the assignment statement above the subroutine declaration, `something` is treated as a string because at the time of parsing the assignment statement the subroutine is not declared yet. That is also why it is always a good idea to forward declare all subroutines early in your program. Apart from giving you the flexibility of defining the subroutines in any order you prefer, you also help prevent Perl from interpreting subroutine invocations as barewords.

As you can see, a bareword is always subject to two possible interpretations which you cannot explicitly tell just by looking at the syntax. That explains why most programming languages today, especially very structured ones like C and Java, never honour flexibility (or ambiguity) like this. You may rule out the use of barewords that do not refer to subroutines by carrying out the `subs` check. There a a few exceptions, however. Barewords on the left hand side of the => operator is always treated as a string as I mentioned very early in the tutorial when you learned about hash initialization. Also, a bareword appearing inside curly braces {} is also treated as a string, as in the example below:

```
$hash{file}     # $hash{'file'} recommended
```

The `subs` check marks these two exceptional cases as valid use of barewords. However, considering the use of barewords in most cases does not contribute to a major cut to file size, and you are actually making the code more confusing, I do not recommend the use of barewords altogether. That's why barewords are rarely used in this tutorial.

The `vars` check is likely to affect you most among the three if you enable it. If you access a scalar variable, an array or a hash that is not lexical (declared with `my`) and without qualifying it with the package name when the `vars` check is in effect, a compile time error is generated. For a long time, we have been using the shortcut that a package variable without being qualified with the package name refers to the current package that is in force. Therefore, that implies the package `main` unless you have set up additional packages. However, when the `vars` check is in effect, you are forced to refer to lexical and package variables in different ways. The following summarizes the syntax, which has already been covered earlier in this tutorial:

```
package MyPackage;

my $lexical = 4;
local $MyPackage::pkgVar = 10;      # instead of just $pkgVar
```

Special provisions have been made to allow you to refer to package variables `$a` and `$b` without the package name, because they are used by the `sort()` function to refer to the two arbitrary picked items for comparison. This also means if you would like to use this `strict` mode, you should refrain from using these two variables because they are not checked for `strict`ness. Also note that typeglobs are excluded from this check because they cannot be lexical anyway. The rationale behind this `strict` mode is to cleanly separate the syntax of variable accesses with respect to package variables and lexical variables. Without `strict` mode, you are subject to a problem similar to barewords mentioned above, illustrated below:

```
$x = 4;         # package variable
sub proc1 {
    $x;         # package variable
}

my $x = 5;      # lexical
sub proc2 {
    $x;         # lexical
}

print proc1, "\n", proc2, "\n";     # 4, 5
```

A similar example has already been given in Chapter 5 to demonstrate how lexical and package variables are resolved. Without the comments, it is not immediately obvious to distinguish the package variable from the lexical. Worse still, you can have a package variable and a lexical of exactly the same name in the same scope. By replacing the above example with a `strict` mode friendly version the whole picture is easier to identify:

```
$::x = 4;       # package variable
sub proc1 {
    $::x;       # package variable
}

my $x = 5;      # lexical
sub proc2 {
    $x;         # lexical
}

print proc1, "\n", proc2, "\n";     # 4, 5
```

10.7.3 The -T Switch — Enable Taint Checking

Strictly speaking, this is a mechanism to enhance program security rather than one which helps catch programming errors. As we have seen above, data passed into your programs from outside can pose significant threats if they are fed to backticks or functions like `eval()`. Although you may honour users of your programs to exercise self-discipline to specify valid data only, you should not carry this attitude. As demonstrated by the Internet spam mail problem, you can hardly rely on users to use your programs properly. A programmer who are conscious of security issues will instead regard all incoming data as if they are malicious, carry out all necessary validations and accept them only if they are valid. This defensive approach is the basic principle behind taint checking in Perl. Perl marks external data passed into the program as tainted. Before you feed such kind of data to potentially unsafe functions, you are required to manually untaint it. This forces you to think more carefully about whether every piece of input data is safe.

In other words, you have to untaint a piece of data if both of the following conditions are satisfied:

- ⋆ it originates from outside of the program (and is thus tainted);
- ⋆ it, or its derivative, is going to be involved in a potentially unsafe operation. A potentially unsafe operation is one which may potentially change the state of the system.

Taint checking is more of a concern for CGI scripts and scripts which are executed setuid or setgid (see Appendix D for more information about setuid and setgid on Unix systems). In fact, for Perl scripts executed that has the setuid or setgid file permission bit set, taint mode is performed automatically and you do not have to enable it.

Examples of tainted data include command-line arguments, data read from user input, files or received from network functions and environment variables. Results of system calls such as `readdir()` and form data received in CGI applications are also considered tainted.

Examples of potentially unsafe operations are `system()`, backticks, `eval()`, opening a file with writable privilege. Filesystem functions such as `unlink()` and `rename()` and the low-level system

calls that are not covered in this tutorial are also unsafe. You may find that all of them have one aspect in common — they may be potentially used to alter the state of the system. Note that `print()` itself is **not** considered a potentially unsafe operation. If you can `open()` the file with writable permission, then you are allowed to write into it even if data written into it are tainted.

For example, in a program involving the `open()` function in write or append mode, if the filename derives from external data (such as input by user) then you are required to untaint it.

The only way you may untaint a piece of tainted data is to extract from it by means of backtracking with regular expressions. For example, the following example allows you to get the directory listing as generated by the `ls` command of arbitrary directories inside the current directory, but not outside of it.

```
1  #!/usr/bin/perl -Tw
2
3  use strict;
4
5  my $path = $ARGV[0];
6  if (defined $path and $path =~ m/^(((\w|\.\w|\.\.[.\w])[.\w]*\/?)+)$/) {
7      # $1 contains the path
8      $ENV{'PATH'} = '';
9      print `/bin/ls -l $1`;
10 } else {
11     print "Invalid or missing relative path!\n";
12 }
```

In this example we also demonstrated how to get the command line arguments. Command line arguments are passed into a Perl program in the `@ARGV` array. Each command line argument becomes an element of this array. For example, you may try out this program by either of these commands below, depending on your situation:

```
# Directory listing of "images/" subdirectory

./ls.pl images/             # Unix with executable chmod
perl -Tw ls.pl images/      # Otherwise
```

We used the pattern built at the end of the previous chapter to check if the path is a valid relative path. Note that the PATH environment variable is cleared. In taint mode, you are required to set the environment variable PATH in your program to a known value. This is needed if you are to invoke operations that execute another program, such as backticks and `system()`. This restriction is mainly set for applications started from the command line because PATH, just like any other environment variables, can be set by the user who executes the program. Therefore, its content cannot be trusted. This environment variable, as introduced in the beginning of this tutorial, is used to specify a sequence of directories to be searched for executables, so that users do not have to always qualify the executable with the absolute path for convenience reasons. If the attacker prepends to PATH the path of a directory under his or her control, the operating system will first search that directory for executables, followed by the system-defined ones. For example, if the `ls` program above is not qualified with an absolute path, and a malicious version of `ls` is placed in that directory, it will be executed instead. Note that this not only affects the program started from within the Perl program, it affects external applications started by that program as well. With a PATH that is known to be

valid, this attack will not be successful.

It is not necessary for PATH to be empty. For example, the following is still valid:

```
#!/usr/bin/perl -Tw

use strict;

my $path = $ARGV[0];
if (defined $path and $path =~ m/^(((\w|\.\w|\.\.[.\w])[.\w]*\/?)+)$/) {
    # $1 contains the path
    $ENV{'PATH'} = '/bin';
    print `ls -l $1`;
} else {
    print "Invalid or missing relative path!\n";
}
```

You don't have to worry about modifying the environment variable. In fact this change is local only to the Perl program in question and other programs that get started from there. Other parts of your system is not affected.

Note that if taint mode is in effect, `@INC` no longer has the "." entry by default. Therefore, modules in the current directory will not be automatically accessible. This arrangement is mainly introduced to protect Perl scripts that are executed from the command line. In particular, if your script is written to be executed by everyone on the system you are vulnerable to a certain attack if your script has to source in external files while having a relative path like "." in `@INC`. Say you have the following program `myprog` in `/usr/local/bin` which can be executed by all users on your server system:

```
#!/usr/bin/perl -w

BEGIN {
    @INC = (@INC, '../lib');
}

use MyModule;   # in /usr/local/lib
# ...
```

A relative path is one which depends on the current directory. ".." is still a relative path because it denotes the parent directory of the current directory. Using a relative path to locate external modules suffers two problems. The first being the user has to manually change the directory to the directory of the script in order to execute it, for example,

```
cd /usr/local/bin
```

before running the script. Most importantly, this leaves a backdoor for a malicious user to replace the definition of `MyModule` with whatever he or she desires. What he or she needs to do is to run this program from somewhere in his or her home directory:

```
cd /home/cracker/bin
/usr/local/bin/myprog
```

Because the current directory is /home/cracker/bin, perl will try to load MyModule from /home/cracker/lib. What the attacker has to do is to place his or her own version of MyModule there, very likely containing malicious code which will be duly executed by Perl. Note that a malicious user does not have to modify the modules placed in /usr/local/lib, which are not writable for him in order to launch a successful attack.

Even if you prefer not to use taint mode, you are advised to replace all relative paths in @INC with absolute paths to avoid the attack outlined in the above example. There are two possible ways to patch against this security hole. The first way is to replace all relative paths with absolute paths. For example,

```
BEGIN {
    $basedir = '/usr/local';
    @INC = (@INC, '$basedir/lib');
}
```

If taint mode is on, and you need to place some modules in the same directory as the script itself, then do not forget to append its absolute path as well. The second method is to use the chdir function to change the current directory with respect to the script. Then it is still safe to use relative paths in the program. For example,

```
BEGIN {
    chdir '/usr/local/bin' or die "Cannot chdir(): $!\n";
    @INC = (@INC, '../lib');
}
```

It is possible that chdir() may fail. Therefore, to be discreet you may wish to check the return value of chdir() (a false value if failed) and stop the program if it fails.

For CGI scripts, again, because the script is executed by the Web server and the current directory is usually set to the directory containing the script itself, it is generally safe to append relative paths to @INC in CGI scripts. It is still a good practice to use absolute paths, however.

While taint mode is a means to achieve a higher level of security, it is not the end. There are certain kinds of attacks that taint mode cannot protect against, such as reading of privacy-sensitive files (containing password entries, for example), because reading a file does not change the state of the system, and hence is not considered a potentially unsafe operation. For example,

```
#!/usr/bin/perl -Tw

my $filename = $ARGV[0];

open FILE, "</usr/local/mywebapp/$filename" or die "Cannot open file!\n";
```

While you may think the script can only access files inside /usr/local/mywebapp, what an attacker needs to do is to pass a string containing "..". For example, if $filename is ../../../etc/passwd then the attacker will still be able to access /etc/passwd. On some Unix systems, especially older installations, this file contains the encrypted passwords of users on the system. However, security-aware personnel know that even if passwords are stored encrypted it is still possible to

recover them. The algorithm used to encrypt these passwords is well known to be easy to break.

You may prefer to hide or obfuscate your source code and believe by doing so you can achieve a higher level of security. You don't. Remember, *security through obscurity* is not an appropriate attitude towards protecting your system. Once the secret is revealed for whatever reasons, security is gone. The most secure system is one which remains impenetrable even if all of its internals are presented before the attackers. This goal is frankly not easy to achieve, and requires the programmers to stay constantly vigilant over potential ways of abuse. Your level of sophistication in security is very often proportional to your experience in this field. You should also keep an eye on security mailing lists to keep yourself aware of the latest kinds of vulnerabilities and attacks discovered.

Summary

- Fatal exceptions are severe errors which halt execution unless trapped with `eval{}`.
- Warnings are less severe but usually indicate logical errors in your programs.
- You may raise fatal exceptions and warnings manually with the `die()` and `warn()` functions respectively.
- `eval()` can be used for runtime evaluation of a given string as a Perl program.
- `system()` or backticks can be used to execute external programs or shell commands.
- The `Error` module on the CPAN can be used for flexible and convenient exception handling.
  - All exception objects inherit `Error`.
  - The `try` block contains statements to be checked for exceptions.
  - The `catch` block is a handler which handles exceptions of a certain class.
  - The `otherwise` handler is used as a generic handler which handles any kinds of exceptions.
  - The `finally` block is executed if the corresponding `try` block completes successfully or if a handler has been executed.
  - If there is no matching handler in the current environment, the exception traverses up the call stack until an appropriate handler is present.
  - A semicolon is needed after the block of the last handler.
  - An exception object may be manually thrown using the `throw()` method.
- `-w` switch enables warnings on dubious operations.
- The `strict` pragma disables some unsafe constructs in your program.
  - `refs` check disables the use of symbolic references.
  - `subs` check disables the use of barewords.
  - `vars` check requires all non-lexical variables to be qualified with package name.
- `@ARGV` contains the command-line arguments.
- Taint mode serves to enhance program security.
  - Data originating from outside the program is marked as tainted.
  - A potentially unsafe operation may change the state of the system.
  - Tainted data has to be untainted before involving in a potentially unsafe operation.

- The only way to untaint is by means of backtracking in pattern matching.
- The "." directory is absent in `@INC` in taint mode by default.
- The PATH environment variable needs to be set to an untainted value before executing external programs from within your Perl program.

Web Links

10.1 perl.com: Object-Oriented Exception Handling in Perl
http://www.perl.com/pub/a/2002/11/14/exception.html

10.2 perl.com: Beginners Intro to Perl — Part 6
http://www.perl.com/pub/a/2001/01/begperl6.html

10.3 CGI/Perl Taint Mode FAQ
http://gunther.web66.com/FAQS/taintmode.html

10.4 perlsec manpage — Perl Security
http://www.perldoc.com/perl5.8.0/pod/perlsec.html

Chapter 11

CGI Programming

11.1 Introduction

Up to this point we have been writing Perl scripts that are to be executed on the command line. As I pinpointed early in this tutorial, the ability to write CGI programs with Perl is the prime motive behind learning Perl for many people. In the following sections we will first look at what CGI is, and understand how it allows webmasters to create dynamic content. Towards the end, I will introduce security issues concerning CGI scripting.

11.2 Static Content and Dynamic Content

11.2.1 The Hypertext Markup Language

The earliest Web servers only serve **static content** in the form of Web pages, or in a more technical parlance, HTML files. A document written in the **Hypertext Markup Language**, or HTML, is actually a plain text document with extra markup added that indicates the logical structure of the document. For example, consider the following HTML document:

```
<html>
    <head>
        <title>A Sample HTML Page</title>
    </head>
    <body>
        <p>This is a paragraph.</p>
        <img src="logo.gif" width="80" height="60" alt="">
        <a href="http://www.cbkihong.com">This is a hyperlink</a>
    </body>
</html>
```

This is a very simple HTML document. ... denotes the text in between is a paragraph. It is then followed by an image of dimensions 80×60. At last, we insert a hyperlink that, when clicked by a user through a browser window displaying this document, will cause the browser to load the resource at the URI "http://www.cbkihong.com" into the browser window. This is not an HTML tutorial, and if you are not familiar with HTML you should learn it first before proceeding with this chapter. But what I would like to demonstrate here is that a markup language serves to differentiate different elements present in a document. In this example, the browser, having received and parsed the HTML document, finds out that the document consists of a paragraph, an image and a link.

Then the browser knows how to **render** the markup and therefore create the objects specified in the browser window. Because a hyperlink has different properties and actions from an image display, the browser needs a way to figure out the kinds of objects present in the document — and that's what HTML is for.

11.2.2 The World Wide Web

HTML documents are special as they contain **hyperlinks**. Hyperlinks allow readers to jump from one document to another document with a **Uniform Resource Identifier** (URI). In order for a document to be accessible on the World Wide Web, it has to be assigned an address. The URI (a more widely used term is Uniform Resource Locator, URI) is the address in this context. For authors, hyperlinks not only make referencing internal or external destinations more convenient, they also bind these separate documents together in the form of linkages. Therefore, with a single URI to an HTML document a reader not only can have access to the document identified by the URI, but also resources linked to that HTML document. Such linkages bind all the linked resources on the Internet into a virtual network, and this is the **World Wide Web** we are using every day.

The World Wide Web utilizes the **client-server model**. First we need to establish what a **server** and a **client** is. In everyday language, a server usually refers to a mainframe or other powerful computational devices, in contrast to personal computers. However, in Computer Science parlance, server and clients are identified by their roles. A server refers to any entities that provides services to clients. In the client-server model, a client first initiates a request and address it to the server. Upon receipt of the request from the client, the server carries out any necessary actions to fulfill the request, and then return the results as a response to the client. Therefore, typical interactions between a client and server in the client-server model can be visualized as in Figure 11.1.

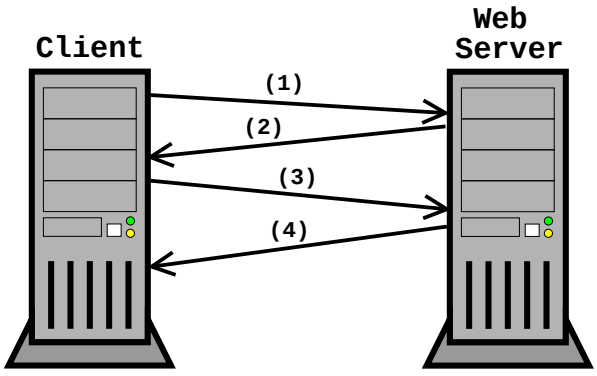

Figure 11.1: A simplified HTTP client-server interaction

Note that in the client-server model, server and client do not necessarily refer to any physical devices. In previous chapters, you have learned how to construct modules that represent objects, or in the procedural approach, represents a set of functions under the same namespace. This situation can fit into the client-server model too. The modules can be thought of as providing services to users of these modules. In this case, the modules act as servers and a program that uses these modules acts as the client.

Have you ever thought about what a Web server is? In fact there is nothing mysterious or compli-

cated. It is merely a system with a suitable **Web server daemon** installed. A Web server daemon is a small program that is executed in the background that handles HTTP requests and responses. A Web browser is actually one of the forms of a **user agent**. When a user enters a URI in the browser and hit the "Go" button, the browser, the user agent in this case, sends a request to the Web server concerned encapsulating the command of getting the resource located at the requested URI. The daemon captures this request, retrieves the specified resource if any and returns the content of which to the client as a response. The browser receives the resource. If it is an HTML document, it parses and renders it so that the document is eventually displayed in the browser window. Rendering refers to the process of converting HTML into the graphical objects displayed in the browser window. The previous figure shown actually illustrates the interaction between the client requesting the HTML document and the server processing the request. The connections in the figure are numbered such that you can more easily refer to the explanation below to understand what each message flow is for.

The interaction between the client and the server is one of the main concerns of this chapter. The client and the server may run on vastly different system architectures. For instance, the Web server may be running on Solaris while the client on Microsoft Windows 2000. However, a common **protocol** defines the common language of communication between the two parties so that platform-independent interaction is made possible. On the Internet, a single protocol is defined for the World Wide Web, which is the **Hypertext Transfer Protocol**, with the more widely known abbreviation of HTTP. Enacted by the **World Wide Web Consortium**, or W3C, HTTP is an open standard that can be freely implemented on any platforms.

Let us briefly outline how documents can be made accessible on the World Wide Web. The administrator of the Web server sets aside a directory (or folder) on the server. All the documents that are to be made accessible on the World Wide Web are placed in this directory (and subdirectories if any). When the URI is received by the Web server, this address is mapped to a location in this directory representing the resource to be retrieved for the client.

(1) represents the initial HTTP request to the Web server. Before that, the Web browser has to accomplish several preliminary tasks. This include transformations of the human-oriented domain name and hostname in the URI to the IP address necessary for the Internet routing system to deliver the request to the intended Web server. After the IP address of the Web server is identified, the HTTP request is encapsulated in a **packet** and delivered to the Web server. A packet is the container of the message. Note that an HTTP request is not sent as is. It is put inside the packet as the payload, and the packet header contains all the necessary information needed to deliver the packet to the destination. This is analogous to a letter being placed into an envelope before it is posted. The HTTP request is similar to the letter, and the envelope with the sender's/recipient's addresses is similar to the packet and its header in this analogy.

Having received (1), the Web server will retrieve the document specified. From the file extension, the Web server recognizes the resource as an HTML document and, therefore, returns the content of the HTML document and marks it as of type "text/html". This is exactly (2).

The client receives the packet containing the returned response. Note that a Web server may return content of types other than HTML. For example, it may be a PDF document or simply some audio clips. Therefore, a means have to be in place that allows the Web browser to identify the type of the returned content. That explains why the returned content in (2) have to be marked of type "text/html". Upon knowing this is an HTML document, the browser would parse it and draws the Web page in accordance with the HTML received. Recall that an HTML document may contain external references (images, audio/video clips, external style sheets or Javascript, Java

applets etc.) that have to be fetched as well in order to display the Web page properly. In the HTML document shown earlier, the image "logo.gif" is the only external reference that has to be fetched. Therefore, a request for this resource is represented by (3). The Web server, on receipt of (3), would return the resource and mark it as of type "image/gif" (4). Note that if the HTML document has multiple external references, additional connections has to be made by the user agent to request such resources. However, usually the user agent will not fetch such external resources one by one. Consider a Web page consisting of 30 images. It would be too time consuming to request each resource sequentially as the network can be slow. Usually, the browser will send multiple requests at a time to the Web server by opening multiple **threads**. The Web server is also likely to be multithreaded to allow it to handle multiple incoming requests concurrently.

A Web server, in this way, serves only static content. With the same URI, anybody would be accessing exactly the same resource at any instance. Also, visitors will not see any differences across visits unless the files have been physically modified. Presentation of static content is generally adequate for many Web sites. However, in order for the World Wide Web to become an interactive media, the ability to serve dynamic content is desired. Dynamic content is usually achieved by writing scripts, especially server scripts. These scripts can generate the output in real time to clients based on input from the clients and data stored in the server database. Therefore, it is possible that every visitor to a Web site serving dynamic content may see different layouts and content customized according to their preferences. That is what that makes the World Wide Web a powerful and interesting media compared with conventional media.

11.3 What is CGI?

In general Computer Science terms, an **interface** provides a well-defined way of interaction between a system and external entities. Recall that in the object-oriented programming paradigm, each class exposes itself to the outside through an interface consisting of methods and properties, and users of the classes do not need to (and should not have to) know the details of the implementation and, instead, access the objects through their interfaces.

In Figure 11.1 we saw the interaction between the Web server and the client. As the resource is static (an HTML document on the filesystem), the Web server can return the specified resources if they exist. However, if the resource specified is an executable script, the script will then need to be executed before returning the generated response to the client.

The **Common Gateway Interface** (CGI) specifies the mechanism through which the Web server should pass data pertaining to the HTTP request to the server script, thus allowing the server script to capture data from the client.

Compared with other protocol specifications, the specification for CGI is intriguingly simple and short that looks more like a tutorial rather than a specification[1]. As you will see, the principle behind the CGI is very easy to understand. Simply speaking, when the Web server receives a request for an executable CGI program, the program is executed. If the program is an interpreted one, it needs to invoke an appropriate interpreter to execute it (which is the perl executable for the purpose of this tutorial). The program writes to the standard output, and the content of which is then returned to the client.

[1] Some efforts of transforming the CGI specification into a formal specification with well-defined grammar is ongoing. Visit http://cgi-spec.golux.com for details.

NOTES

Note that not all Web servers have CGI script execution enabled. In fact, the administrator of the Web server needs to explicitly enable execution of CGI scripts, set up a CGI script **handler** and associate file extensions of CGI scripts permitted (e.g. .pl and .cgi) with it such that the scripts will be executed instead of being fetched and displayed in the client's browser window! Execution of CGI scripts pose certain levels of security risks, and are especially dangerous if either the scripts contain a lot of security vulnerabilities or the Web server is improperly configured. Therefore, many Web hosting companies do not allow execution of CGI scripts in free accounts. Some Web hosting companies set up separate filesystems for hosting CGI scripts to prevent damages due to CGI script attacks or malfunctioned CGI scripts from affecting the entire filesystem. Later in this chapter I will introduce some techniques to write more secure Perl programs to be deployed as CGI scripts.

In order for a CGI script to have access to certain information that are only known by the Web server, such as the remote IP address of the client and the languages the client supports (which is important to certain sites which serve content with multiple language versions) that are only available in HTTP request headers, the CGI stipulates the Web server to make available such information by setting additional environment variables before the Web server executes the CGI program. Figure 11.2 illustrates this point.

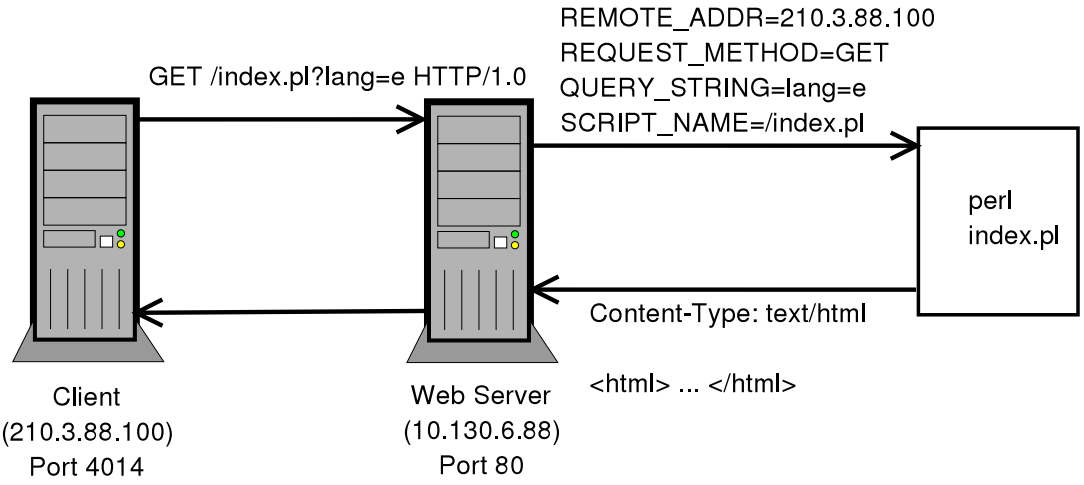

Figure 11.2: CGI script execution

The figure shows the process of a client requesting the CGI script "index.pl". The command displayed is the actual HTTP request being sent by the browser to the Web server, the syntax of which complies with the HTTP specification. `GET` is the action that indicates the method used to send the request. We usually use `GET` or `POST` to send HTML form data to the Web server. Following the action is the path to the resource requested. The browser removes the domain name or IP address from the URI, as it is not used by the Web server to get the specified resource. At the end, the HTTP version is specified, and in this diagram, HTTP/1.0. Notice the string after the question mark in the path. These are passed on the URI, and the Web server extracts the text after the question mark (if any) until whitespace is encountered. These are parameters and are made available by the script by setting the `QUERY_STRING` environment variable as indicated in the diagram. The diagram shows

several important environment variables that are usually set and their corresponding values. Note that some other environment variables are set as well, but the diagram would be too large to fit in and are thus omitted.

As the script is being executed, it writes to the standard output as the CGI response. Upon completion of execution, the Web server collects the CGI response and returns them to the client, inserting any HTTP response headers necessary at the top to comply with the HTTP specifications. In the diagram, it is assumed the content returned is an HTML document, and the type of which is indicated on the first line of the response. This would become part of the HTTP return header. HTTP headers are read and recognized by the browser but hidden from users. The HTTP specification stipulates a blank line between the header and the content. Therefore, you should put a blank line after the last line of the HTTP header, and no blank lines should be present before that.

There is a common misconception by many people, especially for those who are not familiar with the CGI mechanism, to think Perl is CGI, or vice versa, which is highly erroneous. As I have tried to explain previously, CGI itself is the mechanism that allows executable server programs to access information pertaining to the HTTP request through standardized means such as environment variables, instead of being a programming language. Therefore, it is a fallacy to use the terms "Perl" and "CGI" interchangeably. In fact, any programming languages may support the CGI mechanism. CGI programs are not confined to interpreted languages like Perl or Python etc., compiled languages like C/C++ may also be used to develop CGI programs that are used on a Web server. Therefore, the term "CGI script" is not adequately specific, although most CGI programs are written in interpreted languages, preferably Perl. Therefore, it is discreet to refer to an executable CGI-enabled Perl script as a "CGI Perl script". However, for the purpose of this chapter, I would use the term "CGI script" to refer to a "CGI Perl script", for simplicity.

11.4 Your First CGI Program

Having understood all the necessary concepts you need to know for CGI programming, it's time to get your feet wet by building your first CGI script in order for you to understand how a typical CGI script is constructed.

In this section we are going to build an HTML form that contains a textbox for the visitor to input his/her name, and a "Submit" button. After the user has pressed the submit button on the form, a CGI script will be invoked that prints a phrase of greeting based on the time of the server. Therefore, both the form and the script needs to be written.

EXAMPLE 11.1

HTML Form (greeting.html)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<html>
    <head>
        <title>Greetings</title>
    </head>
    <body>
        <form action="greeting.pl" method="post">
```

```
            <p style="font-weight: bold">Please enter your name:</p>
            <input type="text" name="name" maxlength="30">
            <input type="submit" value="Submit">
        </form>
    </body>
</html>
```

Greeting Script (greeting.pl)

```
#!/usr/bin/perl -w


use CGI;
use CGI::Carp "fatalsToBrowser";

$obj = new CGI;
$params = $obj->Vars;

$visitor = $params->{'name'};
@timeFields = localtime time;
$hour = $timeFields[2];

print "Content-Type: text/html\n\n";

if ($hour < 12) {
    $greeting = 'Good morning';
} else {
    $greeting = 'Good afternoon';
}

if (!defined $visitor or $visitor !~ /\w/) {
    $visitor = 'visitor';
}

print qq~
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<html>
    <head><title>Greetings</title></head>
    <body>
        <p>$greeting, $visitor!</p>
    </body>
</html>
~;
```

In order to execute the CGI script, you will need a Web server account that allows execution of CGI scripts. There are a couple of free Web hosts on the Internet that lets you deploy self-written scripts. I have used and am satisfied with the service of Spaceports, but you may find better bargains elsewhere. Such free CGI hosts are excellent places for you to become familiar with CGI Perl programming before you are sophisticated enough to set up your own Web server for development or get more powerful hosting package with paid hosts. If you would like to set up your own Web

server for testing, please flip to Appendix-C for installation and configuration instructions.

Upload both files to your Web server account. Put the script in the same directory as the form. Please note that some Web servers set aside a "cgi-bin" directory inside your account where CGI scripts are only allowed to be executed inside. For some accounts, CGI scripts may be placed and executed anywhere in your account. This is subject to the server configuration and you should consult the system administrator for details. Anyway, in short, put both files in the same directory where CGI script execution is allowed.

Next, you will need to give your files the correct access permissions if your account is on Unix-variant systems. Changing access permissions is commonly known as "chmod". Most probably you would be using FTP (File Transfer Protocol) clients to upload your files to your Web server account. You will need to use an FTP client that supports chmod, like WS-FTP, leechFTP, SmartFTP on MS Windows systems. If you are using Linux with X-Windows installed, most probably you may want to check out gftp or kbear. If you have telnet/SSH access to your account or you have direct access to your Web server filesystem, you may chmod on the command line too, but I'm not going into details here as they are very basic skills Unix users should have already been familiar with.

The chmod values to give to each file and their corresponding verbose representation is:

```
greeting.html: 644 (rw-r--r--)
greeting.pl: 755 (rwxr-xr-x)
```

Now you are ready to test your script. Enter the URI of greeting.html in the address field of the Web browser and press Enter. Please check with your system administrator or relevant instructions from the hosting service on the URI to use. The form should be loaded. Enter your name and click "Submit", and a message will be displayed if there aren't any errors.

Figure 11.3: The "greeting" script in action

Both the HTML form and the CGI script are indeed very simple. Compared with earlier scripts executed on the command line, several elements are new. First, we have used the `CGI` module to fetch the HTML form data in the form of name-value pairs to the CGI script. Next, on line 12 we `print` a line containing the content type information. The "real" server response in HTML format is between line 24 and 33.

On line 5, we `use` the package `CGI::Carp`. By importing the "`fatalsToBrowser`" symbol, this package automatically generates an error page with the error string and debugging information such as the filename and number of the line on which the error occurs whenever a runtime error occurs. Simple CGI programs usually do not perform extensive exception handling and simply use this package to generate the error page. However, as I mentioned in the previous chapter that these error messages are seldom useful to CGI script users who are not the developers, sophisticated scripts usually have their own exception handling routines builtin such as what I presented in the previous chapter.

The `CGI` module is the preferred way for a Perl 5 CGI script to handle CGI-related operations. There are two major operations a CGI script needs to handle in particular. It needs to check if there are any incoming data being passed to the script. Such data are usually passed to the script as parameters on the URI (the GET method) or as form data (the POST method). The script is then executed, and results from execution have to be returned to the client through the Web server. The HTTP user client (presumably the Web browser) captures the response and renders it in the proper way as previously mentioned. In this example, incoming parameters are fetched by the CGI module and returned as a hash reference (`$params`) by the `Vars()` object method. This is usually the most convenient way to get all the incoming parameters in a single operation. The module actually parses the CGI environment variable `QUERY_STRING`, which can be obtained by `$ENV{'QUERY_STRING'}`. Many Perl programmers (and some Perl 5 book authors alike) tend to write their own code of getting HTML form data. An example is quoted below:

```
1  if ($ENV{'REQUEST_METHOD'} eq 'POST') {
2      read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
3  } else {
4      $buffer = $ENV{'QUERY_STRING'};
5  }
6
7  @pairs = split(/&/, $buffer);
8
9  foreach $pair (@pairs) {
10     ($name, $value) = split(/=/, $pair);
11     $value =~ tr/+/ /;
12     $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
13     $FORM{$name} = $value;
14 }
```

The problem is evident. It requires too much typing, and you are actually reinventing the wheel. Because `CGI` is now a standard module bundled in every Perl distribution, there is hardly a reason for not using it. Also, the code snippet above is not tolerant against malformed URIs or alternative URI formats. For example, some scripts use the new convention of separating key-value pairs by `;` instead of the widely-accepted `&`. Unless you are very familiar with such alternative formats, many of which are not well documented in standards or specifications, your code will fail with such URIs. On the other hand, the `CGI` module was carefully developed and has been under constant scrutiny by the Perl community to recognize as many alternative formats as possible. As a result, it is more

reliable and well-maintained, not to count its easiness of use. It is instructive to understand how we can get the form data from the environment variables though because in this way you would understand more about the CGI specification, yet you are not recommended to get the form data manually in your production code.

Recall that earlier I mentioned the Web server would include the content type in the return response. Line 12 is doing exactly that. For static content, the Web server knows the type of content being returned. However, this does not hold for CGI scripts as it is the CGI scripts that decide on the content to return. Therefore, the script should supply this piece of information by the `Content-Type` header field. `Content-Type` is one of the most frequently used declaration in HTTP headers. HTTP headers have to be `printed` before the real content, and an empty line should exist between the header lines and the content. Web browsers look for the header-content boundary in this way so that it can hide the headers properly from users. As no more headers are specified after the content type line in this example, the extra empty line is produced by two consecutive `\n`.

A CGI script does not have to generate all necessary HTTP headers because the Web server should generate them automatically. An exception is `Content-Type`, for the reason that I have just explained. Apart from defined HTTP headers your scripts may generate other custom headers you prefer. However, because CGI scripts are usually rendered in browsers, unrecognized headers are generally ignored, anyway.

11.5 GET vs. POST

In the previous example, we have used a form to transmit user-specified data to the server CGI script. Note that we have used the GET method to pass the form data, as characterized by the `method` property of the `form` element. However, another method POST is also available. Here we shall discuss what they are and how they differ from each other.

To start with, let's investigate how the form data are transmitted in each case. Both methods involve construction of a query string of the following format:

name1=*value1*&*name2*=*value2*&...

Recall the HTML form listed in greeting.html. <form> ... </form> encloses the form. Inside the form, we can find a text entry control, whose `name` attribute is "name". Each control in the form should have the `name` atttribute set. When data is transmitted to the server script, as there may be several pieces of data sent in the same form, every piece of data has to be labeled with a name so that the server script can differentiate the values. The name is set as the `name` attribute in the corresponding form control. In this example, as there is only one control, there is only one name-value pair in the form. Expressed in the above format, that is:

```
name=Bernard
```

What if the name or value contains the characters & or =? In fact, non-alphanumeric characters in the name or the value is encoded, in a manner compatible to RFC2396 for a reason that is to be explained shortly. This document documents the format of Uniform Resource Identifiers, of which Uniform Resource Locators form a subset. The URI encoding scheme stipulates all characters except alphanumeric characters and a few unreserved characters (section 2.3) should be encoded, and this is achieved by representing the character by the hexadecimal representation of its ASCII value preceded by the character "%". Examples:

```
Bernard%20Chan          # Bernard Chan
100%25                  # 100%
A%26B%20Associates      # A&B Associates
```

However, as the space character frequently occurs in data to be encoded, there exists an alternative representation of the space character by the character "+". Therefore, if you are parsing the form data manually yourself (which you are discouraged to do so as explained earlier) you should ensure "%20" and "+" are both treated as the space character. This also explains why it is desirable to use CGI.pm, as to decode properly is not a trivial affair in itself. By using CGI.pm, you don't even have to care such details — it is handled for you automatically.

Having examined the encoding mechanism used on form data, it's time to look at how form data is sent using methods GET and POST.

In the GET method, the query string is appended to the end of the script URI, separated by the "?" character. The form data is part of the HTTP command. That explains why the URI encoding scheme is used for the construction of the query string. An example of the exact HTTP command:

```
GET /temp/greeting.pl?name=Bernard+Chan HTTP/1.1
```

followed by a series of request HTTP headers. These headers together with the values are made available to the CGI script by setting environment variables as well. The environment variable name is the corresponding header field name with the prefix "HTTP_" and all hyphens replaced by underscores. For example, the server script can retrieve the value of User-Agent in the HTTP header by querying the environment variable HTTP_USER_AGENT.

In the CGI specification, it is explicitly mentioned that the value of the `QUERY_STRING` environment variable be set to the query string, that is `name=Bernard+Chan` in this example and the CGI application should parse this variable to retrieve the form data. Note that by using the GET method, beccause the query string is embedded as part of the URI, it would also be displayed in the URI in the HTTP response.

On the other hand, by using the POST method the query string is not embedded in the URI. The exact mechanism is a little bit more complicated. An example of the HTTP command is:

```
POST /temp/greeting.pl HTTP/1.1
```

again, followed by a series of HTTP request headers. However, this time we have an additional line at the end:

```
Content-Type:  application/x-www-form-urlencoded
```

This indicates an HTML form is going to be sent to the server script. The browser then sends the following line:

```
Content-Length:  17
```

This indicates that the forthcoming form data in the form of a query string is of length 17 bytes. The server script can get the content length by querying the environment variable `CONTENT_LENGTH`. The query string is then sent to the server script. In the CGI mechanism, the query string is input to the server script via the standard input (STDIN), and the content length serves as an indication of the number of bytes to read.

You should now be able to understand fully the code snippet for parsing form data in section 10.4. However, as always, use the `CGI` module whenever possible.

So you may ask, should you use GET or POST? My suggestion is to use POST exclusively for transmission of forms. That is, if you have an HTML form that requires your visitors to fill in, use POST. Recall that in the CGI mechanism form data received from GET are saved in the `QUERY_STRING` environment variable. It is known that certain shells may pose a limit to the maximum size of environment variables. Therefore, it is possible that very long forms are truncated as a result. POST does not have this problem because form data, having received by the Web server is directly piped to the standard input from which CGI applications can read. GET is also discouraged because of security issues associated with it. Please read Section 11.9 for security advice concerning the use of the GET form transmission method.

However, the GET method carries a unique characteristic that makes using it unavoidable in some situations. In fact, when you click on a hyperlink in an HTML document, you are actually using the GET method to access it. Because the query string is directly embedded in the URI, when the URI is accessed by your visitor, the query string is sent to the server script automatically without the need of creating an HTML form. This property is vital to CGI applications. Today, some Web sites no longer use static HTML documents to serve its content, but to generate the pages dynamically using dynamic scripting. Usually, on these sites there is a single script which is invoked and the page to view is passed to the script as a query string. The script finds out which page to display by parsing the query string, and the corresponding page is generated and returned to the client. For example, a guestbook CGI application may have one script file `guestbook.pl`, and the various functions are differentiated just by the query string. To read the guestbook you may go through a URI like `guestbook.pl?page=view`, while to sign it the URI is `guestbook.pl?page=sign`. Therefore, you may reach different parts of the CGI application simply by varying the query string in the URI. This function cannot be accomplished with the POST method.

11.6 File Upload

An HTML form is quite powerful in the sense that it also allows Web-based upload of files to the server. If you have used Webmail (Web-based email) services you may already have the experience of affixing attachments when composing an email. This makes use of the file upload capabilities of HTML forms. `CGI` provides certain facilities for you to write Perl CGI programs that accepts uploading of files from HTML forms.

NOTES

Note that many Web servers are configured to disable uploading of files. This is especially the case for free Web hosting services. This is partly because to allow file upload exposes the system to certain security risks. Free Web hosting services usually have a large number of users, and without provisions of remuneration providers of such Web hosting services do not have adequate incentive and staff to carry out all necessary regular security audits and bear the potential costs of intrusions should security attacks occur. Please read section 11.9 for details on the security issues associated with execution of CGI scripts.

Because the details pertaining to file uploading with the HTTP protocol is rather complicated, I will

not go into details the underlying mechanism but to introduce to you the constructs used in this example.

EXAMPLE 11.2 File Upload

HTML Form (upload.html)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN
        "http://www.w3.org/TR/html4/loose.dtd"">
<html>
<head>
    <title>File Upload</title>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body>
    <form enctype="multipart/form-data" method="POST" action="upload.pl">
        <p><b>Please specify a file to upload</b></p>
        <p><input type="file" name="filename" size="60"></p>
        <input type="submit" value="Upload">
    </form>
</body>
</html>
```

File upload script (upload.pl)

```
#!/usr/bin/perl -Tw

use CGI;
use CGI::Carp "fatalsToBrowser";

my $cgiobj = new CGI;
my $fn = $cgiobj->param('filename');        # filename
my $fh = $cgiobj->upload('filename');       # filehandle for reading
my $rInfo = $cgiobj->uploadInfo($fn);
my $byteCount = 0;

print qq~Content-Type: text/html\n\n
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN
        "http://www.w3.org/TR/html4/loose.dtd"">
<html>
    <head>
        <title>Upload Results</title>
        <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
        <style type="text/css">
            .Error, .Normal {font-size: 1.5em; font-weight: bold}
            .Error {color: #FF0000;}
        </style>
    </head>
    <body>
~;

```

```
27 if (!$fh || $cgiobj->cgi_error) {
28     # upload error occurred
29     print qq~<p class="Error">Error: Upload failed</p>~;
30 } else {
31     $fn =~ /^.*?([^\/\\]+)$/;
32     my $outname = $1;
33     open OUTFILE, ">data/$outname";
34     binmode OUTFILE;        # On some platforms, ensure binary file output
35
36     while (my $bytes = read($fh, $buffer, 1024)) {
37         $byteCount += $bytes;
38         print OUTFILE $buffer;
39     }
40
41     close OUTFILE;
42
43     print qq~\t\t<p class="Normal">Upload Successful.</p>\n~;
44     print "\t\t<div>File size: $byteCount bytes</div>\n";
45     print "\t\t<div>File type: $rInfo->{'Content-Type'}</div>\n";
46 }
47
48 print qq~
49     </body>
50 </html>~;
```

On line 7 and 8 we get the filename and the filehandle from the CGI object respectively. In earlier sections I introduced to you the `Vars()` method to get a reference of the hash containing the form data (field, value) pairs. However, there is also the method `param()` that you can use to query the value of a particular field. On line 7 the filename is passed as the value whose field name is "filename", as we specified in the HTML form. The `upload()` method with the field name passed as parameter returns a filehandle from which you may read the uploaded content. This method returns `undef` if the filehandle cannot be read. While uploading a file the browser usually sends along some additional information, such as the content type, in the headers. We can use the `uploadInfo()` method returns a reference to a hash containing this information. You have to pass the filename as the parameter because an HTML form may have multiple file upload fields. On line 45 we try to retrieve the content type using this method. The name of the header is `Content-Type`.

On line 27 we called the `cgi_error()` method to determine whether any CGI errors had occurred. CGI errors rarely occur in practice, unless, for example, file upload dies in the middle as a result of the client stops sending data. The second condition we test is the filehandle being `undef`. This condition rarely occurs, either. If either of these conditions is satisfied, we should consider the upload a failed one. However, please note that the `CGI` module fails to notice if the user keys in a filename which points to a non-existent file in the file location entry.

Line 31 extracts the filename portion from the filename passed in. Note that the filename does not take on any definite formats, and is determined by the client browser. Some browsers send the full absolute path to the script (e.g. on Windows it looks like `C:\somedir\file.zip`), while some other browsers submit only the filename. Note that because the path, if one exists, depends on the client system, we should treat both \ and / as pathname separators. Line 34 enables binary mode, as explained in Chapter 8. We read the file received in 1024-byte chunks and write them to the output file.

11.7 Important Environment Variables

In this section, a few other important environment variables that are useful and are made available through the CGI mechanism are outlined below.

11.7.1 CGI Environment Variables

`REMOTE_ADDR` is set to the IP address of the client sending the HTTP request to the server. This is usually the address of the client machine itself. However, presence of intermediate **proxy servers** between the client and the server may result in seeing the address of the proxy server instead of the client machine itself. This is because the proxy server, on receipt of the client HTTP request, replaces the source address with its own before sending it to the server. Proxy servers are set for various purposes, such as caching and imposing security control. `REMOTE_HOST` is the fully qualified domain name (hostname and domain name) of the machine identified by `REMOTE_ADDR`. As the machine concerned may not have a domain name, this variable may be NULL. You may also access it through a `CGI` object by invoking the `remote_host()` method.

11.8 Server Side Includes

One of the major reasons why many people consider PHP more convenient than Perl is that you can embed PHP code inside parsed HTML documents. On servers that support it, there exists a feature that is known as **Server Side Includes** (SSI). SSI refers to a set of directives that are placed in an HTML document, and evaluated when the document is parsed by the Web server. The directives are replaced by the result of evaluation.

The capabilities of Server Side Includes is very limited. Among the several functionalities supported, the most commonly used feature is to embed the results of a CGI program in an HTML document. The SSI directive used is the `include` directive, which looks like this:

```
<!--#include virtual="counter.pl" -->
```

where `counter.pl` is the CGI program to be executed. Note that only a file path is supported, it cannot be a URI. The following shows an HTML page with this SSI directive and the source of the program `counter.pl`:

EXAMPLE 11.3 Counters (Server Side Includes)

HTML Test Page (testpage.shtml)

```
<html>
    <head>
        <title>Test Page</title>
    </head>
    <body>
        <p>This is a test page.</p>
        <hr>
```

```
        <div style="font-size: 0.8em; font-style: italic">This page has been ¶
           accessed <!--#include virtual="counter.pl"--> times.</i></div>
    </body>
</html>
```

Counter script (counter.pl)

```
1  #!/usr/bin/perl
2
3  use Fcntl ':seek';
4
5  print "Content-Type: text/html\n\n";
6
7  my $path = "data/counter.dat";
8
9  # Create if not yet exist
10 if (! -f $path) {
11     open LOG, ">$path";
12     print LOG "0";
13     close LOG;
14 }
15
16 # If cannot be opened, return a '?' for display
17 if (!open(LOG, "+<$path")) {
18     print "?";
19     exit 0;
20 }
21
22 chomp($num = <LOG>);
23 truncate LOG, 0;
24 seek(LOG, 0, SEEK_SET);
25 defined($num) or $num = 0;
26 ++$num;
27 print LOG "$num";
28 close LOG;
29 print $num;
```

Note that `counter.pl` is still executed as a CGI program. Therefore, the program still have to be given an executable chmod and it should output the content type header. Whenever you load the page `testpage.shtml` the counter stored in the data file will be updated and you will see at the bottom of the page the number of views of the current page.

11.9 Security Issues

CGI and Perl together provides a great deal of flexibility and ease in developing Web-aware scripting solutions. However, the issue of security is usually overlooked in the script development process either because the developers are not aware of security issues associated with server-side scripting, or in order to meet completion deadlines all necessary security audits are unfortunately bypassed. The consequence is that a large number of functional scripts but full of security holes are being

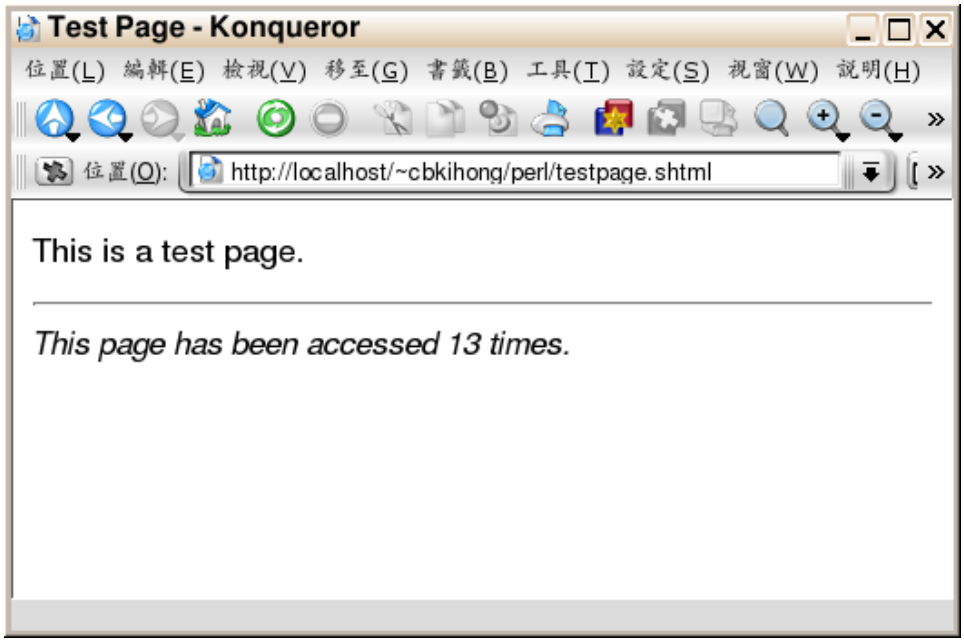

Figure 11.4: Counter Embedded in HTML With SSI

pushed into the market every day, and these programs are introducing new security backdoors to systems on which they are being deployed. It is easy to write a program that is functional, yet many security holes are so subtle that they are difficult to be discovered and thus avoided. In this section, we attempt to highlight certain varieties of attacks possible and compile a set of crude but useful guidelines that would hopefully help you write more secure CGI scripts.

11.9.1 Why Should I Care?

Perl programs are distributed in complete source code, making it very easy for potential crackers to study your source code, locate vulnerabilities and plan for potential exploits. On the other hand, natively compiled programs can only be examined in assembly language and it is notoriously difficult to trace the instruction flow, not to count locating vulnerabilities precisely from them which is a practically infeasible task. Attacks on compiled programs are usually either discovered on an *ad lib* manner or are attacked in general ways by, say, passing a very long sequence of characters to overflow the input buffer (many programs written in C or C++ are vulnerable to this attack). Attacks on interpreted languages are generally more well crafted and precise because the source code is available to target at specific vulnerabilities.

CGI scripts are installed on Web servers and are thus open to visitors anywhere over the Internet. They are available for attack 24 hours a day, 365 days in a year. For standalone systems or systems in an intranet, attacks are only possible from a limited subset of users. On the Internet, attacks may originate from anywhere in the World, hundreds of miles away.

Many CGI scripts are used in business settings or are directly involved in electronic commerce, such as shopping cart programs. The ability to maintain a high level of security has always been paramount in business applications. That is because data handled by such applications very likely include highly-confidential personal data of the clients. Such kinds of sensitive data include the PIN of clients, their record of transactions and personal information like address or social security number etc. Failure to do so not only invites embarrassment to the financial institution(s) concerned,

such institutions may also be held legally liable for improper handling of personal data. Occasional cases have been heard in that the credit card information of celebrities were being captured by crackers through some illegal means and subsequently such data were posted to the World Wide Web, putting the subjects concerned in profound embarrassment. People are unlikely to have confidence in a company being unable to keep such data from being improperly manipulated. This directly leads to loss in revenue.

11.9.2 Some Forms of Attack Explained

HTML Form Tampering

In 2000, Internet Security Systems, Inc. released a security alert on 11 shopping cart applications that were found to be vulnerable to this attack (please visit http://www.iss.net/issEn/delivery/xforce/alertdetail.jsp?id=advise42 for the full text). This vulnerability vividly demonstrates the ignorance of the script developers concerned in security issues of CGI programming.

Shopping cart scripts are set up by merchants on Web servers to track down items users have selected to purchase, usually while browsing online product catalogues. When a user browses the product catalogue and locates an item of interest, he or she clicks on a button or icon to save the product information, usually the product identifier and other items necessary to identify the user such as the username, to the server database. When he or she has finished adding all the items intended to purchase, he or she checks out and the shopping cart is displayed. A shopping cart is an abstraction of the list of items a user intends to purchase, and is private to the user concerned, in a sense similar to a typical shopping experience in modern supermarkets (dropping items into a shopping cart and check out at the cashier). Shopping cart scripts in general implement this by querying the server database with the username and extracts all items to purchase specific to the user. The results are displayed in the form of an HTML form for the user to confirm the order and optionally enter additional information required to process the order, such as delivery address, contact phone number, payment card details etc. (but such kinds of information are generally saved as part of the user profile at user registration so that such details no longer need to be specified every time he or she purchases) An HTML form that is vulnerable to this attack may look like the following:

```
1  <html>
2      ...
3      <body>
4          ...
5          <form action="checkout.pl" method="put">
6              <input type="hidden" name="username" value="bernardchan"></input>
7              <input type="hidden" name="total" value="560"></input>
8
9              <!-- A list of items to purchase, omitted here -->
10
11             <p>After you have confirmed your order, click on the button below.</¶
                   p>
12             <center><input type="submit" value="Submit Order"></input></center>
13         </form>
14     </body>
15 </html>
```

Line 6 and 7 shows two hidden fields in the HTML form. Unlike the visible form controls, such as the text messages and the submit button, hidden fields are not rendered in a browser window. They are typically used to hold additional data which are not bound to any visible controls but required by the server script (the shopping cart script) to process the form input. Notice the `name` and `value` attributes of the hidden fields. The name-value pairs of hidden fields are also sent to the server script along with that of other visible controls when the form is submitted. In this example, the shopping cart script adds the two hidden fields dynamically in generating the HTML form to hold the username and total amount so that on submit the shopping cart software would arrange the total amount indicated to be charged on the customer's credit card account.

And problem comes. Shopping carts with this vulnerability do not verify the ordering information on submit and blindly assumes the value of the hidden fields are exactly as generated by the script itself. Therefore, a user may give himeself or herself a discount by replacing the total amount with a smaller one. Why is this possible? As the form is generated and sent to the user, the browser renders it in the browser window. However, a person with little knowledge will know one can browse the HTML source and even save it as a disk file. What he or she needs to do is to use a text editor to modify the value, save it and load the modified version in the browser. A click on the submit button is all it takes to complete this attack. Because the script does not perform the check on the total, the attack is successful. This attack may only be discovered one day, possibly in year-end auditing, that the payment amount and the order do not match, but that would be too late — the subject may have already closed the account and hid up that you could no longer find him or her anymore.

Why did the script developers commit this error in the first place? Possibly the culprit is "convenience", or "laziness". It takes quite many steps to generate the shopping cart HTML form, and the script developers could make their lives easier by putting all necessary details to process the order on the form instead of having the checkout script calculate the total amount again, because the calculated total needs to be displayed on the HTML form anyway. By means of "hidden" fields they probably thought no one would bother to read the HTML source and discover this security hole. Most people probably would not bother to, but it still leaves a backdoor for abuse. As an e-commerce application I believe having a backdoor like this is totally unacceptable, however pretty or sophisticated the application can be.

Conclusion Do not trust anything sent over the network. Always carry out all verifications possible at the best of your knowledge before committing anything.

Privacy Issues

While this is not directly related to Perl CGI programming in general, you should be aware that the Hypertext Transfer Protocol (HTTP) that is used for accessing the World Wide Web conveys all its messages in plaintext. Therefore, when you submit a form with your password, address or credit card number etc. filled in these fields are all transmitted in plaintext through the Internet. The Internet is a gigantic interconnected network of computers. When you send a message to a remote host, for example, to browse a certain Web page on a remote Web server, the Internet routing system has to find a path between you and the remote Web server. The scale of the Internet is so large that usually you have to go through many intermediate hosts on your way to the remote host. Any of these hosts is able to read the content of the messages, or even to modify it. Under certain circumstances, other malicious hosts may also be able to **eavesdrop** the traffic through these hosts. Therefore, there is no confidentiality at all.

If you Web site has to collect privacy-sensitive information from your users by means of forms, the Web server concerned should be configured to serve these forms using the **Secure Socket Layer** (SSL) protocol or **Transport Layer Security** (TLS), a close variant of SSL. The Internet uses a layered architecture. SSL acts between HTTP on top and TCP/IP, the Internet delivery system at the bottom. Therefore, messages between a Web browser and the Web server are encrypted in transit and thus malicious hosts, even intermediate hosts on the path are unable to decrypt the messages except the communicating parties. For example, when your browser sends a form with your personal information which is to be sent over SSL, the HTTP message encapsulating the form data are passed to the SSL layer which encrypts it and then passed to the TCP/IP subsystem to send it to the remote Web server. The Web server on the remote end reverses these steps to receive the message from the TCP/IP subsystem, decrypts it and then recover the original HTTP message. Configuration of SSL/TLS is performed at the Web server and no modification to your CGI programs is needed.

Earlier in this chapter I mentioned the two form submission methods, namely GET and POST. In the GET method, the form data encoded as a query string are also carried on the URI. This poses a number of security issues if such form data contain privacy-sensitive information. First, many browsers now cache recently visited URIs so that users can easily revisit them without having to manually bookmark them or otherwise save them. This causes potential privacy violations on those single-user systems (and thus no password is asked to log on the system) or misconfigured multi-user systems. For instance, consider a typical patronage to a cybercafé in your neighbourhood. You logged on a PC inside and conducted an online transaction to buy something from an online store. Suppose the GET method was used and thus form data including your password in plaintext were carried on the URI, and was thus cached by the browser. Then you logged off and left the café. Then another person logged on the PC, and when he or she started the browser, a URI to the online store with your password embedded appeared in the browsing history. Does this sound scary to you? System configurations of most public workstations generally should have this issue fixed already. However, this is not a guarantee. Another situation where this URI may be divulged is due to a header field in HTTP that is called "Referer". When a user agent, e.g. browser, sends an HTTP request message to the remote Web server to retrieve a certain resource, the user agent may include in the header field "Referer" the referring URI, that is, the URI of the document from which the URI of this resource was obtained. For example, when you are viewing a Web page on a certain site like `http://www.somesite.com/links.html` and you click on one of the hyperlinks there to `http://www.anothersite.com`, for example, the URI that refers to `links.html` may be included in the HTTP header which is sent to `anothersite.com`. Therefore, if the referrer URI refers to a script URI with privacy-sensitive information embedded in it they will also be carried along with the next URI access.

You, as a CGI script and Web developer, may do your part to protect your customers by using the POST form transmission method instead of the GET method whenever privacy-sensitive form data are involved because form data are not carried in the URI. That does not eliminate the need for encrypted tunnels such as SSL to keep out of prying eyes over the network, however.

`eval()` and Related Attacks

As we have seen in the previous chapter, `eval()` and backticks are very dangerous and frequently taken advantage by attackers to execute on the Web server whatever commands they specify. As noted, enable taint mode to check for potential insecure input data that may be passed to potentially dangerous operations. Construct precise patterns to ensure all input data match the patterns intended. Do not use these operations unless that is absolutely necessary. Also bear in mind that taint mode is not the end. There are a lot of traps taint mode cannot guard against.

Because all of these points have been duly introduced in the previous chapter, I am not repeating it here again.

11.9.3 Safe CGI Scripting Guidelines

Today, new CGI scripts emerge on the market every day. However, how many of them are really secure? A script that has a pretty interface does not imply it must be well tested for security bugs. Because most users of CGI scripts are not programmers, they cannot protect themselves but rely on you, the script developers, to safeguard your programs against abuse. Therefore, you have an irrefutable obligation to ensure your programs are as secure as possible.

Below I put together a set of guidelines that you can check against your programs to evaluate if they are secure. Some of the points have been discussed earlier in the tutorial but being repeated here as a summary. These guidelines are not exhaustive, however. Despite the section heading, these guidelines pertain to any kinds of programs receiving input from outside of the program. Discreet security personnel are well-educated about these guidelines. So should you.

- ⋆ Refrain from `eval()`, backticks and `system()` whenever possible. If you do use it, check its content. Ensure the content is exactly in the form expected and enable taint mode.
- ⋆ For incoming data, it is generally more secure to define a pattern that describes what is allowed than what is disallowed. For example, to check if an email address is syntactically valid you create a pattern which describes the syntax of a valid email address, for example `/^\w+@(\w+\.?)+$/`. With this deny-everything-by-default arrangement, an email address that matches this pattern must be a syntactically valid one. This is preferable to a pattern which enumerates the invalid characters and test whether the string contains these invalid characters, because given the large number of possible characters you may not be able to enumerate them.
- ⋆ Do not be more permissive than necessary. That is, do not set a file permission value to more than what you need to get your program running. More permissions imply higher risk.
- ⋆ Implement **rings of security** where applicable. A single line of defense is generally weak because if this line of defense is broken, intrusion will be successful. In security analysis, we try to locate all possible **attack paths** in the program. An attack path describes a sequence of steps an intruder may take to launch a successful attack. To implement rings of security implies we should try to establish multiple lines of defense on these paths — even if a line of defense can be broken into, there is still another barrier that deters attackers from proceeding. The more barriers to surmount, the less likely an intrusion attempt will be successful and, very likely, the attacker will give up and turn to another more vulnerable system or program instead.

Summary

- A document written in HTML is a plain text document with extra markup added that indicates the logical structure of the document.
- The World Wide Web is a content presentation and distribution platform built on top of HTTP user agents and Web servers.

- HTTP is the protocol used for World Wide Web access. A protocol defines a common language of communication between parties.
- A Web server serves static content by retrieving resources from the filesystem in the form of files and return them to the client side.
- A Web server may also serve dynamic content by executing some programs on the server and return the output generated to the client side.
- A Web browser is just an HTTP user agent which initiates HTTP requests, and captures responses from Web servers and renders them to produce the graphical display as Web pages.
- CGI, the Common Gateway Interface, defines the method of interaction between the Web server and an executable server program.
- CGI stipulates that Web servers should pass certain information pertaining to the HTTP request received from the client side to server executables by means of environment variables. Generated output are collected by the Web server and returned to the client side, generating any missing HTTP headers as necessary.
- The `CGI` module may be used for a wide variety of CGI-related operations, including form-based file uploading.
- The `Vars()` object method of `CGI` returns a reference to a hash containing incoming form data.
- A Perl CGI program is required to generate the `Content-Type` HTTP header indicating the kind of data returned to the client side.
- In the GET form transmission mechanism, form data are appended to the URI as parameters. In the POST mechanism, on the other hand, form data are sent independent of the URI and CGI programs may obtain them by reading from the standard input.
- The GET mechanism is not recommended because of limitations imposed by the shell and potential privacy breaches through carrying privacy-sensitive data over to third-party servers by means of the `Referer` HTTP header.
- In both mechanisms, form data are transformed into a query string with special characters encoded before being sent.
- Server Side Includes is a mechanism that you may use to embed the generated output of a CGI program in a server-parsed HTML document. Usually, server-parsed HTML documents take on the extension `.shtml` instead of `.html`.
- Server CGI scripting may easily expose a server to security breaches if CGI scripts are not written in a careful manner.
- Do not trust anything sent over the network, including form data received from the client. Carry out all verifications at the best of your knowledge before accepting them.
- HTTP messages are sent in plain text. Security-aware Web applications should serve their content over encrypted tunnels such as SSL or TLS to protect their users.
- Refrain from runtime evaluation and system shell access for CGI programs if possible. Otherwise, enable taint mode and verify input data to ensure they are safe.

Questions

A. We have seen how a shopping cart application with the form tampering vulnerability could be abused to give adversaries discounts on items paid for. In the make-up HTML form example shown, identify another attack that may be performed and devise a mechanism

to defend this attack. Assume on submit that the value received through the "total" field is charged on the user's account whose name is "username" immediately without performing any verifications.

Appendix A

How A Hash Works

A.1 Program Listing of Example Implementation

This appendix is a general, language-independent overview of how a hash table (or simply a hash) works. I will present the complete source code of a Perl class module `Hash`, which is a hash implemented in Perl. All basic hash operations have been built into the module. After you have learnt how a hash table works, you can then implement one likewise in another programming language, in case one is not available. This implementation is written in an object-oriented style so that it can be reused and extended very easily. Understanding of the source program shown below requires familiarity of object-oriented programming in Perl. Therefore, you should be able to understand the whole of the source code if you have read through this tutorial and, as a result, I will not attempt to describe the programming constructs in detail in the text below.

Please note that the given `Hash` class is intended for illustration purpose only, and is not optimized in any way for performance or utilization. Therefore, you are not advised to use it in production environments.

EXAMPLE A.1 Hash Table Implementation

```
1  package Hash;
2
3  # Hash.pm
4  # An example demonstrating how a hash works
5
6  # This hash is not designed for production use and is not
7  # optimized. The Perl builtin hash should be used instead.
8
9  # Number of slots should preferably be prime.
10
11 use strict;
12
13 sub new {
14     my $arg0 = shift;
15     my $cls = ref($arg0) || $arg0;
16     my $this = bless {}, $cls;
17     $this->initialize(@_);
18     return $this;
19 }
```

```
20
21  # Store property values
22  sub initialize {
23      my $this = shift;
24      my %params = @_;
25      foreach (keys %params) {
26          $this->{$_} = $params{$_};
27      }
28
29      # Hash defaults
30      !exists $this->{'NUMSLOTS'} and $this->{'NUMSLOTS'} = 17;
31  }
32
33  # Add a (key, value) pair to the hash
34  # i.e. $hash{$key} = $value;
35  sub add {
36      my $this = shift;
37      my ($key, $value) = @_;
38
39      # First call the hash function to find where it should go
40      my $slot = $this->hashFunction($key);
41
42      if (my $cur = $this->search($key)) {
43          # The specified key exists, replace current
44          # value with the new one
45          $cur->{'value'} = $value;
46      } else {
47          # Prepend to the chain
48          my $next = $this->{'slots'}[$slot];
49          $this->{'slots'}[$slot] = {
50              'key' => $key,
51              'value' => $value,
52              'next' => $next,
53              'prev' => undef,    # must be head
54          };
55          $next->{'prev'} = $this->{'slots'}[$slot];
56      }
57  }
58
59  # Remove a (key, value) pair from the hash
60  # i.e. delete $hash{$key}
61  sub remove {
62      my $this = shift;
63      my $key = shift;
64
65      # First call the hash function to find where it should go
66      my $slot = $this->hashFunction($key);
67      if (my $toDelete = $this->search($key)) {
68          my $prev = $toDelete->{'prev'};
69          my $newNext = $toDelete->{'next'};
70          if (defined $prev) {
```

```
71              $prev->{'next'} = $newNext;
72          } else {
73              $this->{'slots'}[$slot] = $newNext;
74          }
75          $newNext->{'prev'} = $prev;
76      }
77  }
78
79  # Search for a specific key in the hash; Internal use only
80  sub search {
81      my $this = shift;
82      my $key = shift;
83
84      # First call the hash function to find where it should go
85      my $slot = $this->hashFunction($key);
86
87      my $count = 1;
88      my $totalNum = $this->getAllKeys();
89
90      my $ptr = $this->{'slots'}[$slot];
91      while (defined $ptr) {
92          if ($ptr->{'key'} eq $key) {
93              $this->{'DEBUG'} and print STDERR "Number of comparisons: $count/¶
                    $totalNum.\n";
94              return $ptr;
95          } else {    # advance
96              $ptr = $ptr->{'next'};
97              ++$count;
98          }
99      }
100     $this->{'DEBUG'} and print STDERR "Number of comparisons: $count/$totalNum.\¶
            n";
101     return undef;
102 }
103
104 # Get value for a specific key
105 # i.e. $hash{$key}
106 sub get {
107     my $this = shift;
108     my $key = shift;
109
110     # First call the hash function to find where it should go
111     my $slot = $this->hashFunction($key);
112     if (my $obj = $this->search($key)) {
113         return $obj->{'value'};
114     } else {
115         return undef;
116     }
117 }
118
119 # Tests if a specific key exists,
```

```
# because get() cannot handle the case when the value is undef
# i.e. exists $hash{$key}
sub exists {
    my $this = shift;
    my $key = shift;

    return ($this->search($key)?1:undef);
}

# ================================================================
# These three functions let you get a list of keys, values or both

# keys %hash
sub getAllKeys {
    traverse($_[0], 'key');
}

# values %hash
sub getAllValues {
    traverse($_[0], 'value');
}

# @list = %hash
sub getAllKeyValuePairs {
    traverse($_[0], 'value', 'key');
}

# Traversing the hash returning values of certain fields
# Internal Use only
sub traverse {
    my $this = shift;
    my @fields = @_;
    my @retval = ();

    foreach my $slot (0..$this->{'NUMSLOTS'}-1) {
        my $ptr = $this->{'slots'}[$slot];
        my @tmpList = ();
        while (defined $ptr) {
            push @tmpList, @$ptr{@fields};
            $ptr = $ptr->{'next'};
        }
        push @retval, reverse(@tmpList);
    }
    return @retval;
}

# Hash function
# Generates mapping (Key -> hash slot index)
sub hashFunction {
    my $this = shift;
    my $key = shift;
```

```
171     my $tmp = 0;
172     for(my $i = 0; $i<length($key); $i++) {
173         $tmp += (ord(substr($key, $i, 1)) * ($i*2+11));
174     }
175     return $tmp % $this->{'NUMSLOTS'};
176 }
177
178 # Total number of slots in this hash
179 sub getTotalSlots {
180     my $this = shift;
181     return $this->{'NUMSLOTS'};
182 }
183
184 # Total number of OCCUPIED slots in this hash
185 sub getOccupiedSlots {
186     my $this = shift;
187     my $count = 0;
188     for (0..$this->{'NUMSLOTS'}-1) {
189         $this->{'slots'}[$_] and ++$count;
190     }
191     return $count;
192 }
193
194 # Return a similar string as
195 # scalar %hash
196 sub getUtilization {
197     my $this = shift;
198     return $this->getOccupiedSlots() . '/' . $this->getTotalSlots();
199 }
200
201 1;
```

In Chapter 3 I tried to compare the efficiency of searching for an item in an array by various methods, namely linear search and binary search. I mentioned that although you may search for an item in an array, you should not try to do so because neither method is efficient when the population (number of items) becomes large. I purported that searching in a hash is more efficient compared with an array. However, I could not give you any evidence for it because you cannot look at the internals of Perl builtin hashes to examine the process of searching. Now this self-written implementation lets you examine the process in more detail. By putting the module in "debug" mode, some extra information reflecting the internal state will be sent to the standard error. In this implementation, when the `search()` internal method is invoked, the number of comparisons will be output to let you see how many comparisons is needed to arrive at the conclusion of whether a hit or a miss occurs.

This example program uses the self-implemented hash. A hash is created and populated with 100 (may be less, if collision occurs, i.e. generation of the same value multiple times) integers and the user inputs a number to check if it appears in the generated list of integers:

```
1 #!/usr/bin/perl -w
2
3 # Search for an element in a hash
```

```
4
5  use Hash;
6
7  $h = new Hash('NUMSLOTS' => '19', 'DEBUG' => '1');
8
9  # Generating 100 integers
10 $NUM = 100;
11 $MAXINT = 5000;
12
13 srand();
14
15 print "Numbers Generated:\n(";
16 for $i (1 .. $NUM) {
17     $valueToInsert = sprintf("%d", rand(1) * $MAXINT);
18     $h->add($valueToInsert, 0); # in fact, any values can be assigned here
19     print $valueToInsert;
20     print ", " unless ($i == $NUM);
21 }
22 print ")\n\n";
23
24 print "Slot utilization: ", $h->getUtilization(), "\n\n";
25
26 print "Please enter the number to search for >> ";
27 chomp($toSearch = <STDIN>);
28
29 # Hash search here
30 if ($h->exists($toSearch)) {
31     print "\"$toSearch\" found!\n";
32 } else {
33     print "\"$toSearch\" not found!\n";
34 }
```

A.2 Overview

A hash table is actually composed of arrays which possess additional instruments to ensure efficient indexing of data. A hash table has three fundamental operations — insertion, searching and deletion. They can all be performed very efficiently regardless of the population.

Conceptually, a hash table consists of a number of **slots**. Each hash table is associated with a **hash function**, which transforms the key into a slot index.

To insert a key-value pair into the hash, a slot index is computed from the key, using the hash function. In this way, the key will be assigned to one of the slots. The key-value pair will be appended to the slot.

It is possible that multiple keys will be mapped to the same slot index. We describe this situation as a **collision**. This must occur if the population is larger than the number of slots. However, because of the characteristics of the hash function, it is possible that collision occurs even if the number of slots is larger than the population. Several **collision resolution** mechanisms exist that may be used to address this problem. The method that is described in this section is known as **chaining**. That is,

items that are mapped to the same slot are linked together, like a chain.

To locate an item with a particular key in the hash, we essentially repeat the procedure by computing the slot index. Then, we perform a linear search on the chain attached to the slot. If an item with this key exists in the hash, it must exist in this chain. If we traverse the chain and cannot find a matching key, then we conclude the key is not in the hash. Therefore, with the help of a slot index we reduce the traversal from all items to just items whose key maps to a single slot index. That truly explains why searching is very efficient for hashes.

Other hash operations have to depend on searching. For example, to delete a hash element simply compute the slot index and search for the key in the chain corresponding to the slot index. If one is found, remove the item and fix the linkages in the broken chain.

A.3 Principles

Let the number of slots in a hash be m and the population be n. Therefore, in the ideal case when the population is evenly distributed over all available slots, the average number of items on each chain (**load factor**) is thus n/m, assuming $n \geq m$. If $n < m$, then in the ideal case all m items are distributed over n of m slots, so some slots are empty while the others have the maximum chain length of 1.

Efficiency of hash access is determined by how well (uniformly) the items are distributed over the available slots. Therefore, the hash function used exerts a direct impact. Construction of good hash functions is a topic under constant research. However, how well a hash function performs is not only determined by the hash function itself, but also by the nature of the keys, which depends on the domain in which the hash is to be used. It is probable that a hash function that performs well in one scenario does not perform well in another scenario. Therefore, a perfect hash function can hardly be achieved in practice if the application domain is not known in advance.

An optimal choice of the number of slots is also important. For hash item access, once the slot index is computed, a linear search for the key on the chain attached to the slot has to be performed. Therefore, the shorter is the chain, the more efficient is the hash access. Again, the optimal number of slots also depends on the application domain.

A.4 Notes on Implementation

In the sample implementation, the default number of slots is 17, but you may override it by specifying the `NUMSLOTS` key in the class constructor. The Perl builtin hash is different from this implementation in that the implementation provided by Perl is a **dynamic hash table**. That means, the number of slots is not fixed throughout its lifetime. A dynamic hash table is able to increase the number of slots as items are constantly added to the hash. This keeps the chains short on average so that access is still efficient when the population grows large. However, because the principles of dynamic hash tables is complicated, you should seek external documentation for more information.

The chains in my implementation are constructed in the form of **linked lists**. This is usually the preferred way of implementing a chain in programming languages supporting some sort of references. However, you may also use an array to represent it in case references are not supported

in certain programming languages.

Web Links

A.1 perl.com: How Hashes Really Work
http://www.perl.com/pub/a/2002/10/01/hashes.html

Appendix B

Administration

In this appendix, you would learn how to:

- ⋆ Install a Perl Module from the CPAN

B.1 CPAN

Perl has a very active user community. This is evinced by the gigantic list of modules available on the Comprehensive Perl Archive Network (CPAN). CPAN is the central warehouse where you can search for existing Perl modules other Perl programmers have contributed to the community. You will be surprised that the CPAN contains modules of virtually any category you can think of. By using existing modules on the CPAN you can enforce code reuse and cut down both development time and cost by not *reinventing the wheel*. Even if you are not writing programs in Perl, for example, when you are a system administrator at a web hosting company offering Perl-enabled hosting packages, you still need to know how to install Perl modules your users may need.

B.1.1 Accessing the Module Database on the Web

You can browse the module list at http://www.cpan.org/modules/01modules.index.html. However, in my opinion the best way to locate modules is to use the module search engine. There are several engines available on CPAN, and I generally use http://search.cpan.org because the interface is more neatly, if you know part of the module name already. If you don't know the name of a module and would like to search through the module description, then http://kobesearch.cpan.org would be more useful to you, and it is actually more powerful. You may download the source packages and read the documentation online. If you intend to install the modules in the traditional way (but manually), you may also download the source packages there.

B.1.2 Package Managers

Perl Package Manager

As most Windows systems are not equipped with a suitable compiler suite (for example, Microsoft Visual C++), and some Perl modules contain portions written in C for performance improvements (see below), Perl modules are usually distributed in the form of packages. In case a module has to be compiled before use, it is compiled before being put into the package. The packages are constructed by volunteers who have the compiler to compile the modules, and the packages

generated are then contributed to the community. Therefore, from a user's point of view this is very convenient as all he or she needs to do to install a module is to fetch the package and install it, and it's then ready for use.

Activestate Perl (most likely on the Windows platform) distribution comes with a package manager PPM that I found to be quite convenient to work with. Through this package manager, packages can be automatically fetched from a remote server and installed. It also includes tools to keep your modules up to date.

To start PPM, select "Perl Package Manager" from the "ActiveState ActivePerl 5.8" program group on the Start menu. You would see a prompt `ppm>` which gives you a command-line interface to type your maintenance commands. Don't be frightened by a command-line interface in case it looks awkward to you (this is a GUI age, after all). It is very easy to use and you can always access help information by typing `help` alone. To get help information on module upgrades, type `help upgrade` etc.

To start with, you would first try to search for modules in the module repository. Try to type in `search Crypt` to see a list of cryptographic modules available. By default, PPM would searches for modules by name only. You can use logical operators `and`, `or` and `not` in the query string. To match both the module name and description, use the command `search Crypt or ABSTRACT='Crypt'`. To install a package, note the package name and use the command `install`, for example `install Crypt-TripleDES`. The latest stable version of the `Crypt::TripleDES` module is automatically downloaded and installed. To keep your modules constantly up to date, type `upgrade * -install` to upgrade any packages that are out of date to the latest version. You can browse a list of packages installed by `query *`.

This section is meant to give you an overview of the most commonly performed operations with PPM. Please read the documentation bundled with ActiveState ActivePerl for further information. However, because not all modules are already packaged for PPM, you may not be able to use this method for some modules.

B.1.3 Installing Modules using CPAN.pm

If your Perl distribution does not come with a package manager like PPM, as is the case for most Unix variants, there is still an easy way to compile and install modules on the CPAN without resorting to the traditional, but manual method (to be described next). It is to use the CPAN.pm module that is bundled with your Perl distribution (should be, but please check it for sure). This module gives you a convenient way to automate the regular extract-configure-compile-install steps. It does not give you as many features as PPM, but it is more than adequate if you would just like to install a module in an easy way. However, a new module called CPANPLUS.pm is going to replace CPAN.pm in future Perl releases, offering a few more features. You may wish to use it instead of CPAN.pm. However, as of this writing it is not bundled in perl so you have to install it separately. The instructions below apply to CPANPLUS as well, but please replace CPAN with CPANPLUS.

You can use CPAN.pm in two ways. If you have tried PPM above, you can also have a shell-like command line interface where you can type the maintenance commands. Alternatively, to quickly install a module you may not wish to go into the CPAN.pm shell and you can simply type it on your system command line. Both methods are covered below. To start the CPAN.pm shell, type `perl -MCPAN -e shell` on the command prompt. Note that very likely you need to be the system administrator (root) in order to start the CPAN shell.

```
cbkihong:~# perl -MCPAN -e shell

cpan shell -- CPAN exploration and modules installation (v1.76)
ReadLine support enabled

cpan> h

Display Information
 command  argument          description
 a,b,d,m  WORD or /REGEXP/  about authors, bundles, distributions, modules
 i        WORD or /REGEXP/  about anything of above
 r        NONE              reinstall recommendations
 ls       AUTHOR            about files in the author's directory

Download, Test, Make, Install...
 get                        download
 make                       make (implies get)
 test      MODULES,         make test (implies make)
 install   DISTS, BUNDLES   make install (implies test)
 clean                      make clean
 look                       open subshell in these dists' directories
 readme                     display these dists' README files

Other
 h,?           display this menu       ! perl-code   eval a perl command
 o conf [opt]  set and query options   q             quit the cpan shell
 reload cpan   load CPAN.pm again      reload index  load newer indices
 autobundle    Snapshot                force cmd     unconditionally do cmd
```

To install a module, for example, `Crypt::DES`, you can use the command `install Crypt::DES`. The module would be downloaded from a local package repository, extracted, compiled, tested and finally installed on your system. Meanwhile you would see a lot of messages printed on the screen. Errors, if any, will also be printed. Therefore, if the process suddenly stops in the middle and there are some errors printed, you should try your best to locate the source of the error. Usually, it may be that you have certain prerequisites not met yet. You should look at the documentation that comes with a module if problems arise to see if they are addressed in there. If you don't want to use the CPAN shell, you can specify the command on the command line, for example, `perl -MCPAN -e install Crypt::DES`.

Apart from modules, you can install **bundles** through the CPAN module. A bundle is a set of related modules that are packaged together. Usually a module requires a set of other modules, and in this case, the packager may prefer to package all of them as a bundle. An example of a bundle is `Bundle::DBD::mysql`, which includes the modules `DBI`, `DBD::mysql`, `Data::ShowTable` and `MySQL`. Each bundle has a bundle file which lists the modules covered. The `autobundle` command of your CPAN shell lets you create a bundle file which lists the modules that are currently installed on your system. For example, if you are a system administrator for a Web host you can maintain a current list of modules on a single system and have the other systems share the bundle file to ensure that all servers are installed the same versions of Perl modules. For more information, please see the CPAN manpage.

B.1.4 Installing Modules — The Traditional Way

The traditional way is to download the compressed source package from the CPAN as mentioned above, and extract it. This method should in general only be used if the CPAN.pm module isn't or cannot be set up properly. The packages are compressed with gzip, a popular compressing tool on the Unix platform. You can extract the package by the command `tar xvzf` *`package.tar.gz`*. If your version of tar(1) does not support the 'z' switch, you will need to try `zcat` *`package.tar.gz`* `| tar xvf -`. On Windows you can decompress gzipped tarballs with software like Winzip or PowerArchiver. After the package is extracted, change to the directory containing the extracted sources on the command line using the `cd` command. Now the Makefile needs to be created. Run the Makefile generation program by `perl Makefile.PL`. The program detects the settings of your Perl installation and creates the site-dependent Makefile (that explains why the Makefile is not included in the sources). To execute the Makefile, you need a make tool which coordinates the whole compilation session. On most Unix systems you should have a version of make already available. If you are on Windows, you need to have Visual C++ installed for compilation. nmake is included in the Visual C++ installation. Because nmake may not be on your PATH, you may need to use the Visual Studio command prompt which adds the necessary paths to the PATH so that the necessary tools can be invoked without specifying path. On Unix, type `make`; while on Windows `nmake`. The files would then be compiled. Then type `make install` or `nmake install` to copy the necessary files to the correct location. Note that modules installed in this way are not recognized by any package managers.

Although Perl source files are generally portable, some modules have to make use of the XS mechanism to delegate part of the program in C for performance considerations, or when they need to interface with the system native libraries in order to function. That's why compilation is sometimes needed. If compilation is needed you need to have a working installation of the C compiler available. For ActiveState Perl the compiler required is `cl.exe` of Visual C++, and gcc on Unix platforms. If you don't have a compiler, you can still install modules which do not use the XS mechanism (that is, there are no .xs files in the bundle). On Windows, nmake is free and you can download from ftp://ftp.microsoft.com/Softlib/MSLFILES/nmake15.exe which is a self-extracted executable.

Web Links

B.1 perlmodinstall manpage — Installing CPAN modules
http://www.perldoc.com/perl5.8.0/pod/perlmodinstall.html

Appendix C

Setting Up A Web Server

You have learned how to develop CGI scripts with Perl. It is not very practical to test CGI scripts with the Perl interpreter alone, because you cannot easily reproduce an environment resembling a real Web server. There are several Perl-enabled free web hosting services on the Internet. You may wish to test your scripts there, or on a paid web hosting account that you own. However, that is still inconvenient having to upload and test every time you would like to test your scripts. Also, using a third party server for script testing is dangerous. First, if your script goes astray and locks itself in an infinite loop (this is far more easier to occur than you think), it would become a never-ending process that sits there wasting CPU time and system resources. Every invocation of the script concerned increases one faulty process. It does not require many clicks before the server would eventually be brought down. Although you may start to wonder your script has gone into an infinite loop, you cannot do anything because the processes are executed as a system user (on a Unix system, quite likely it's a pseudo user "nobody") instead of you and, even if you have shell access to the Web server, you don't have the privilege to terminate these faulty processes. Helplessly, a few hours you receive a furious letter from the Web host administrator about locking up your account or so. It's not a story. This was a pathetic experience of mine in my early years of Perl script development. Also, don't forget free web hosts usually don't give you access to error logs, so you would get no clue why your scripts do not function as expected. A testing Web server comes to rescue. In fact, you can set up an entirely free Web server in less than half an hour. In this appendix, I would give you some instructions that guide you through setting up a basic Perl-enabled Web server that you can use to test your CGI scripts offline. The instructions were tested on my system, namely Windows 2000 Professional and Debian Linux 3.0 (woody).

C.1 Apache

You can download the latest version of Apache Web server from Apache. For your convenience, the link to the latest version as of this writing is provided below:

Apache 2.0.47 (Windows binary)
Apache 2.0.47 (Unix source)

C.1.1 Microsoft Windows

Note that the installation file uses the Microsoft Installer for installation. If your Windows version is ME or 2000, you are ready to execute the installation program. If you use Windows XP, you need to install Windows XP Service Pack 1 first if you haven't installed it yet. If you have older versions of

Windows, you may need to install MSI Installer. Double click on the .msi file just downloaded. If that doesn't work, you don't have MSI installer set up yet and need to install it first. Please search the Microsoft Web site for the download link.

You should be logged in as a user whose privilege allows installation of software, say administrator. The installation program should start. You should disable any firewall software on your system before proceeding with the installation. Also, if you have any other Web servers running which binds to port 80 you should disable them first. On Windows 2000, go to the Services window. Right click over the entry "World Wide Web Publishing Service" and select "Stop".

After you have agreed to the terms of the license agreement and press "next", specify the network domain, server name and administrator's email address. If a domain name is available on your network, enter it into "network domain". Otherwise, put `localhost` for both "network domain" and "server name" like I do. You can change these settings later on in the Apache configuration file manually, so the settings that I specify in the screenshot is generally adequate.

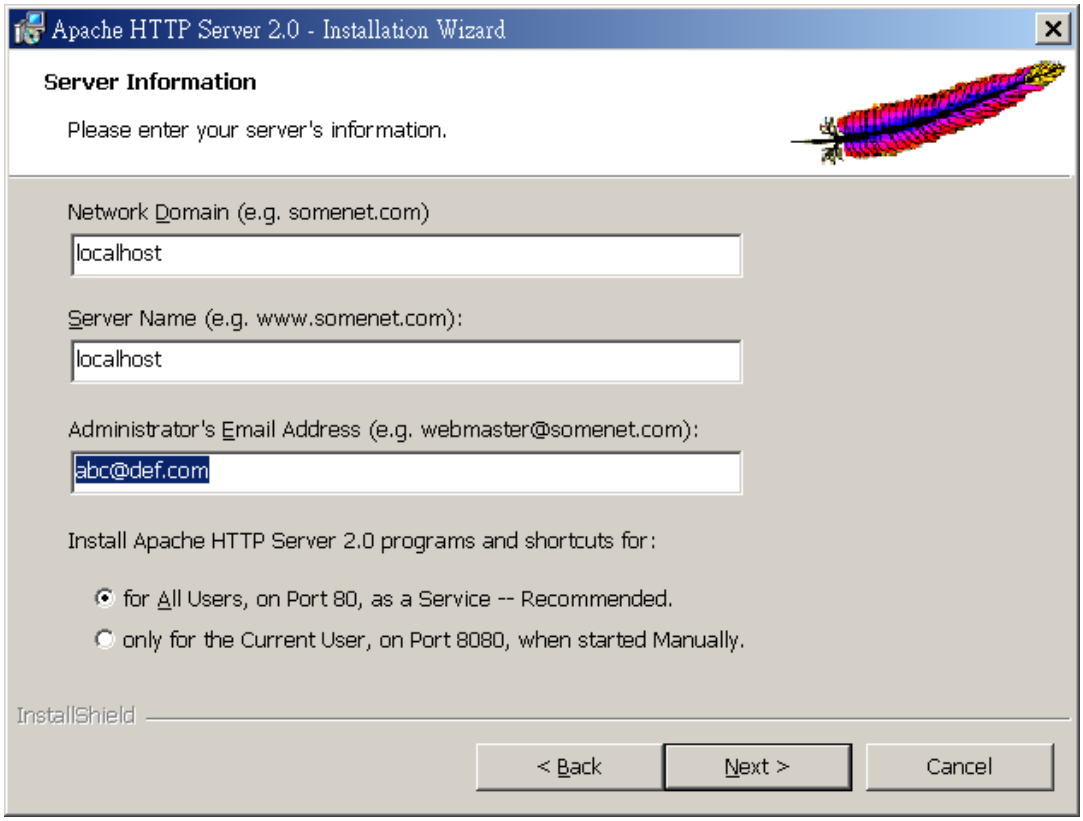

Figure C.1: Apache Installation Options

Then select installation type. Choose "Typical" here which is generally sufficient, unless you would like to compile modules on your own later on (quite rare on a Windows system). Then choose the installation folder. The default value is usually acceptable. Installation then begins. When installation is completed successfully, Apache should have already been started as a service.

Test your Web server by accessing http://localhost. You should see the default Apache test file, as shown (the language of the page may differ, but you should see a similar page with the Apache logo).

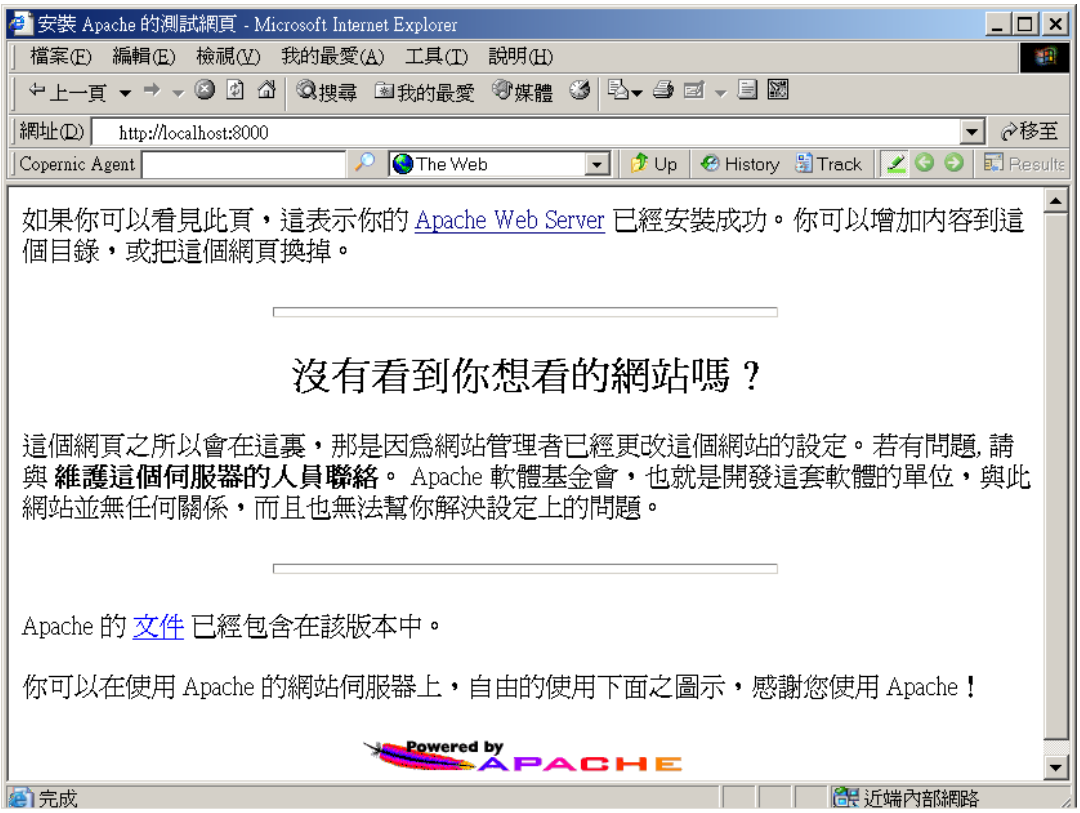

Figure C.2: Apache Default Index File

Now you can edit the configuration file. Go to the Start Menu >> Apache HTTP Server 2.0.47 >> Configure Apache Server >> Edit the Apache httpd.conf Configuration file. Notepad appears containing the configuration file.

If you would like to run IIS and Apache concurrently, search for the two lines like

```
Listen 80
ServerName localhost:80
```

They are at different locations in the file and I just put them here together for convenience. Change the two lines to

```
Listen 8000
ServerName localhost:8000
```

which instructs Apache to bind to port 8000 instead of port 80. IIS still gets port 80. If you don't need (or don't have) IIS, you don't need this step.

Now we add in support for CGI. Find the line

```
#AddHandler cgi-script .cgi
```

and change it to

```
AddHandler cgi-script .cgi .pl
```

That is, remove the # at the front and add ".pl" at the end. This makes Apache recognize this file extension as CGI scripts.

Now enable CGI scripts to execute in a specific directory you prefer. On Windows NT-series (including 2000 and XP) operating systems you are recommended to put your scripts under your own "My Documents" folder for easy access. Very likely you already have a "My Webs" folder in it that has already been created for you by your Windows installation that you can use. Find out a section like

```
UserDir "My Documents/My Website"

#<Directory "C:/Documents and Settings/*/My Documents/My Website">
#    AllowOverride FileInfo AuthConfig Limit
#    Options MultiViews Indexes SymLinksIfOwnerMatch IncludesNoExec
#    <Limit GET POST OPTIONS PROPFIND>
#        Order allow,deny
#        Allow from all
#    </Limit>
#    <LimitExcept GET POST OPTIONS PROPFIND>
#        Order deny,allow
#        Deny from all
#    </LimitExcept>
#</Directory>
```

Remove the # as shown below and correct the path (for example the drive label on my system is F: instead of C:, and the folder containing the scripts and HTML files is changed from the default "My Website" to "My Webs"). Change the "Options" line as shown below, and add the ScriptInterpreter-Source directive, which instructs Apache to find the path to Perl from the Windows Registry instead of from the shebang line:

```
UserDir "My Documents/My Webs"

<Directory "F:/Documents and Settings/*/My Documents/My Webs">
    AllowOverride FileInfo AuthConfig Limit
    Options MultiViews Indexes SymLinksIfOwnerMatch Includes ExecCGI
    <Limit GET POST OPTIONS PROPFIND>
        Order allow,deny
        Allow from all
    </Limit>
    <LimitExcept GET POST OPTIONS PROPFIND>
        Order deny,allow
        Deny from all
    </LimitExcept>
</Directory>

ScriptInterpreterSource Registry-strict
```

We would like to use the Registry to resolve the Perl installation path because most Perl scripts (especially third-party scripts) today simply use the Unix-style `#!/usr/bin/perl` shebang line and in order to use these scripts you will need to correct all the shebang lines to the correct Perl path, which is not convenient. If the Perl path is resolved from the Registry, then the path on the shebang

is ignored. This is generally recommended unless you are not confident with editing the Registry yourself. If you don't feel comfortable editing the Registry, then comment out (put a # in front of) the ScriptInterpreterSource line above. Then, you will need to edit the shebang line of all your scripts to reflect the path to your Perl interpreter, e.g.

```
#!F:/Perl/bin/perl.exe
```

or, if your interpreter is on the PATH, you can use instead

```
#!perl
```

Now save the Apache configuration file and restart the Web server by Start Menu >> Apache HTTP Server 2.0.47 >> Control Apache Server >> Restart.

At last, you will need to edit the Windows Registry. This is dangerous if you did it improperly as proper functioning of your Windows system relies on the integrity of the Registry. Always stop and think before you commit your action because Registry operations are irreversible! To run the Registry Editor, select Start Menu >> Run and in the box type `regedit.exe`, and press Enter.

Data entries in the Windows Registry are arranged in a tree, rather like a directory structure of a filesystem. The left pane contains a tree containing a hierarchy of keys. When you click on a key in the left pane, the right pane contains a list of values associated with that key.

In the left pane, under "My Computer" you would find a number of nodes (keys). In earlier versions of Windows you would not find a "My Computer" root node, but that does not matter. Now expand the "HKEY_CLASSES_ROOT" node and find the ".pl" node below. If Activestate Perl is installed, the key should be there. Then right click on the ".pl" node and choose "New" >> "Key". A new key is created under ".pl". Type "Shell" for the name. Similarly, create "ExecCGI" under "Shell", and "Command" under "ExecCGI". Click on "Command", and in the right pane double click on the text "(Default)" and enter "F:\Perl\bin\perl.exe" "%1", don't forget to update the path to the Perl executable if this is not correct, and click OK.

When finished, the Registry Editor should look like Figure C.3, with all the nodes expanded.

You can now close the Registry Editor. Create a simple Perl CGI script:

```
1  #!/usr/bin/perl
2
3  # You should update the shebang line above if you don't have
4  # the necessary Registry entry or have not added the line
5  # ScriptInterpreterSource Registry-Script to httpd.conf
6
7  print "Content-Type: text/html\n\n";
8
9  print "<html>
10     <head><title>Test script</title></head>
11     <body>
12         <h1>Hello World</h1>
13     </body>
14 </html>";
```

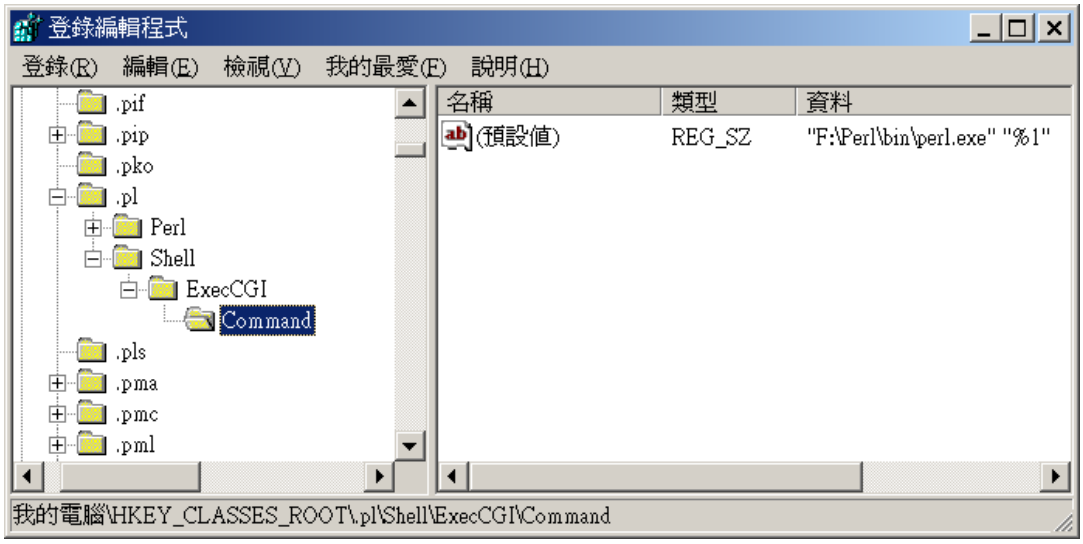

Figure C.3: Registry After Modification

Recall Apache have been configured to allow execution of Perl CGI scripts in the "My Webs" folder in your "My Documents" folder. Save your file there with a ".pl" extension, for example on my system it's F:\Documents and Settings\Administrator\My Documents\My Webs\test.pl.

Now, access http://localhost/~administrator/test.pl. Replace "administrator" with the username. If you have adhered to this tutorial, it would be "administrator". If you have changed the Apache port number to 8000 replace "localhost" with "localhost:8000" in the URL. You should see the "Hello World" text in the browser window. Congratulations! Your Web server has been set up!

If you have disabled IIS temporarily and changed the Apache port number to anything other than 80, you can now restart IIS if you would like to.

C.1.2 Unix

By far the only uniform way to install Apache on nearly all Unix platforms is to install a source distribution. Binaries for different Unix variants are not interoperable, and are installed in a different manner. For example, if you are on mainstream Linux distributions you may be able to find RPMs (RPM Package Manager) compiled for your distribution. Debian Linux also has its own package manager. FreeBSD also has a port system. Therefore, you should possibly check the documentation on your system to see if binaries are available. However, binaries are usually slightly out of date because you need to wait until packagers package a software into binaries suitable for your platform. Installation instructions for binary packages are not described here.

You can extract the package by the command `tar xvzf httpd-2.0.47.tar.gz`. If your version of tar(1) does not support 'z' switch, you will need to try `zcat httpd-2.0.47.tar.gz | tar xvf -`. After the package is extracted, change to the directory containing the extracted sources on the command line using the `cd` command. First, we set up the configuration options of Apache by the `configure` shell script. Use this command:

```
./configure --enable-mods-shared=all --prefix=/usr/local
```

which installs Apache using the installation prefix /usr/local. All modules are enabled and compiled

as shared libraries, instead of linking them into the main Apache executable. In this way, we only dynamically load a module when it is needed. The CGI module is one of the modules that would be compiled. You may specify additional options if you like. Type `./configure --help` for a list of configuration options that can be applied. If you don't see any errors at the end of the messages, then you should be fine. Now compile and install it.

```
make
make install
```

If no error messages appear which causes the compilation to stop, then you are lucky. Now Apache should be installed. Here I outline the changes to be made to the Apache configuration file. If you adhere to this tutorial the configuration file should be at /usr/local/etc/httpd.conf. Because this is similar to that of in the previous section I'm not going to explain the options that have been covered there. This is the original configuration:

```
User nobody
Group #-1

ServerAdmin you@your.address
#ServerName new.host.name:80

#<Directory /home/*/public_html>
#    AllowOverride FileInfo AuthConfig Limit Indexes
#    Options MultiViews Indexes SymLinksIfOwnerMatch IncludesNoExec
#    <Limit GET POST OPTIONS PROPFIND>
#        Order allow,deny
#        Allow from all
#    </Limit>
#    <LimitExcept GET POST OPTIONS PROPFIND>
#        Order deny,allow
#        Deny from all
#    </LimitExcept>
#</Directory>

#AddHandler cgi-script .cgi
```

My modifications are as follows (please read the description below before making your changes):

```
User www
Group www

ServerAdmin abc@def.com
ServerName 127.0.0.1:80

<Directory /home/*/public_html>
    AllowOverride FileInfo AuthConfig Limit Indexes
    Options MultiViews Indexes SymLinksIfOwnerMatch Includes ExecCGI
    <Limit GET POST OPTIONS PROPFIND>
        Order allow,deny
        Allow from all
    </Limit>
```

```
    <LimitExcept GET POST OPTIONS PROPFIND>
        Order deny,allow
        Deny from all
    </LimitExcept>
</Directory>

AddHandler cgi-script .cgi .pl
```

On Unix, although Apache has to be executed as root in order to bind to port 80 (as a side note: you can also install Apache in a user account by modifying the installation prefix `--prefix` during configuration to somewhere inside your home directory but you need to set the port number to be larger than 1024 because port numbers below 1024 are so called **privileged ports** that by Unix convention can only be bound to by the root user), in most installations once Apache has started up it would drop its root privilege through the `chroot()` system call and run as a pseudo user like "www" or "nobody" that has limited privileges to do much harm to the system should intrusions occur. This is a measure to minimize the security risks involved. An appropriate user and group may have been created for you by your operating system installation already. If not, you will need to create them. See the adduser(8) and addgroup(8) manpages for more information. On my system, a "www" user and group has been created and I just use it here.

You should specify the email address of the administrator. In Windows this information is already set up by the installation program, but on Unix you will need to set it here. Similarly is the case for ServerName.

With this configuration, you should be able to execute CGI scripts inside the "public_html" directory. Now start the Apache Web server by

```
/usr/local/bin/apachectl start
```

Appendix D

A Unix Primer

In this appendix, you would learn:

- ⋆ basic concepts of the Unix operating system
- ⋆ some basic commands in Unix

D.1 Introduction

D.1.1 Why Should I Care About Unix?

You may wonder why I took the time to dedicate an appendix to the Unix operating system in a Perl tutorial. "Perl is platform-independent", you said, "and my Perl programs run fine on my Windows XP!" Indeed, Perl programs are largely platform independent (platform-dependent Perl programming is possible, but in most cases you don't need to, so I am not going into details), and you can develop and test Perl programs entirely on your platform. However, a couple of Web server surveys consistently confirm that more than 60% of the Web servers on the Internet run on various flavours of Unix, outweighing Microsoft Windows with the remaining share of about 30%. Very likely, your Web host is also running on Unix. Therefore, it is beneficial to you if you can get yourself acquainted with this operating system.

Also, Perl has a strong Unix culture and tradition. It was intended to be a flexible scripting language like shell scripting, `awk` and `sed` etc. on Unix platforms. For example, regular expressions have long been extensively used in various utilities on the Unix platform. Quite a number of Perl's builtin functions (as seen on the perlfunc manpage) are interfaces to the corresponding Unix commands or closely resemble functions in the standard C libraries on Unix (not to mention the fact that Dennis Ritchie, the inventor of the C programming language, was also one of the inventors of the core Unix operating system!).

D.1.2 What Is Unix?

Unlike Microsoft Windows, which is solely owned and developed by the Microsoft Corporation, the term "Unix" does not refer to any specific operating system releases. It is a collective name embracing a family of operating systems sharing certain common characteristics.[1] It was initially developed by Dennis Ritchie and Ken Thompson from the Bell Labs in 1960s. Over more than 30

[1] The Open Group has issued The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2003 Edition that defines a standard operating system interface, which can be regarded as the basis of Unix operating systems.

years of evolution, many variants of the Unix operating systems have emerged. Today, the most prominent names in Unix include Linux, *BSD (FreeBSD/OpenBSD/NetBSD), Solaris and Mac OS X (Jaguar) as a newcomer.

This appendix is intended to be an introduction to the Unix computing environment. Therefore, emphasis is placed on shell commands that are frequently used by users. In particular, X-Windows, the graphical user interface on Unix, will not be introduced in this appendix. Also, for the sake of generality features or commands that are specific to any particular Unix variant will not be included. The commands and concepts described in this chapter can be applied on many Unix operating systems. Readers are advised to consult other Unix literature for more in-depth treatment of the topics covered.

D.1.3 The Overall Structure

A system running Unix can generally be envisioned as having a layered architecture. What does this mean? First, the bottom layer is the system hardware. This ranges from the central processing unit (CPU), memory system, storage devices, graphics adapter to peripheral devices such as keyboard and printers. They are interconnected through **buses**, that are inter-component wires that carry data and control information in between. The operating system lies on top of the hardware layer to coordinate the hardware components. This part of the operating system which interacts with the hardware directly is the core of the operating system, called the **kernel**. On top of the kernel are the **user programs**. These programs do not need to interact with the hardware directly anymore. Instead, they only need to interface with the kernel in case those services are needed, by invoking the necessary **system calls**, which are functions provided by the kernel. Filesystem operations, for example, are examples of standard system calls on Unix. The kernel would in turn perform the appropriate actions to instruct the devices to commit the operation. When you are using a user program, you are presumably the top among the layers, on top of the user program. The user program provides you with the user interface with which you interact. Commands defined by the user program, and in case of graphical user interfaces, mouse clicks and their corresponding pixel coordinates are translated into instructions for the lower layers. This layered architecture allows a high degree of abstraction between layers. If the interface (or the Application Programming Interface) of a layer changes, only the layer immediately on top needs to be modified. Because of abstraction, the user needs not understand the low-level details of the hardware circuitry, for example, in order to operate a computer system. This is an important principle in all modern computer systems.

To sum up, the Unix operating system serves to act as an intermediary between a computer user and the computer hardware so that activities that occur at different components of a computer system are well coordinated.

Unlike many other operating systems, Unix is a multi-user operating system from the very beginning, allowing multiple users to work on the system concurrently. Multiple users may log on a machine remotely from other machines via the traditional telnet protocol, which gives you remote access to the command-line interface of the machine. Telnet is not secure because data are carried in plain text. SSH (Secure SHell) is a similar protocol but data are encrypted in transit. There is also a protocol called Virtual Network Computing (VNC) that lets users have access to the graphical user interface from the remote side.

D.2 Filesystems and Processes

D.2.1 Overview

In modern days, large-capacity **secondary storage** is important. Floppy disks, hard disks and CD-ROM etc. are classified as secondary storage devices, in contrast to **primary storage**, which is just an alias of the main memory. Secondary storage media is considered permanent, because data that are written onto the media are retained when the power is off. When an executable file has to be executed, the operating system arranges for the executable file to be copied from secondary storage to primary storage (i.e. your RAM). The program is then broken down into instructions and executed from internal caches. A program in execution is called a **process**. The reason that the executable file is not executed from the secondary storage directly is that accesses of secondary storage devices are very slow compared with the main memory because such accesses involve mechanical movement of disk arms, for example, and the speed of which is subject to mechanical limitations.

A disk is simply a large array of disk blocks of fixed size in which data can be stored. It does not mandate any rules to organize data that are being stored on the disk. In order to better organize storage of data, a large-capacity storage media is usually divided into multiple **partitions**. This is very common in systems with multiple operating systems installed, with each operating system installed in its own partition. However, partitioning simply marks the beginning and the end of each partition, but does not answer the need for organizing data that are written in each partition. Therefore, the second step is to create a **filesystem** on each partition. By creating a filesystem, the data structures that are necessary to index data to allow efficient access are written to the partition. This operation is more well-known as "formatting" to users of MS-DOS and Windows operating systems. Modern Windows systems use either one of the two filesystems, namely **File Allocation Table** (FAT) or **New Technology File System** (NTFS). On Unix, various choices of filesystems are available depending on the operating system variant. Among the Unix variants, Linux supports the largest number of filesystems. The standard filesystem on Linux is the **Second Extended Filesystem** (ext2). Other filesystems in widespread use include reiserfs and ext3, which are both equipped with journalling capabilities. Other Unix variants mostly use the **Unix File System** (UFS).

The filesystem determines the directory structure, for example, how files are represented and layout on the disk. To better organize the files, we introduce a hierarchical directory structure. Logically, files are classified and put into different directories. Files of a similar kind are put into the same directory. Also, directories can be nested, so that a directory can be created inside a directory. This allows fine-grained organization of files in a well-structured manner for easy access. Because directories are hierarchical, they are customarily represented in the form of a tree. In Computer Science arena, a tree refers to a hierarchical data structure which best shows the subordinate relationships of directories. This is a very intuitive concept that readers with some experience with operating systems should be familiar with.

The root of the directory structure in Unix is represented by /. The directory root is the only directory which does not have any parents. All files on the system must rest under the directory root. As you can see in the figure above, a number of directories appear under the root. Each of these directories serve its specific purposes. In Table D.1 I list a few more important ones that are present on nearly all Unix systems [2].

In Unix terminology, a file is not necessarily a regular file. Directories, symbolic links and even devices are also represented in the same way as a regular file, differentiated simply by a file type

[2] Please consult the Filesystem Hierarchy Standard, the recommended scheme for compatibility between different Unix variants.

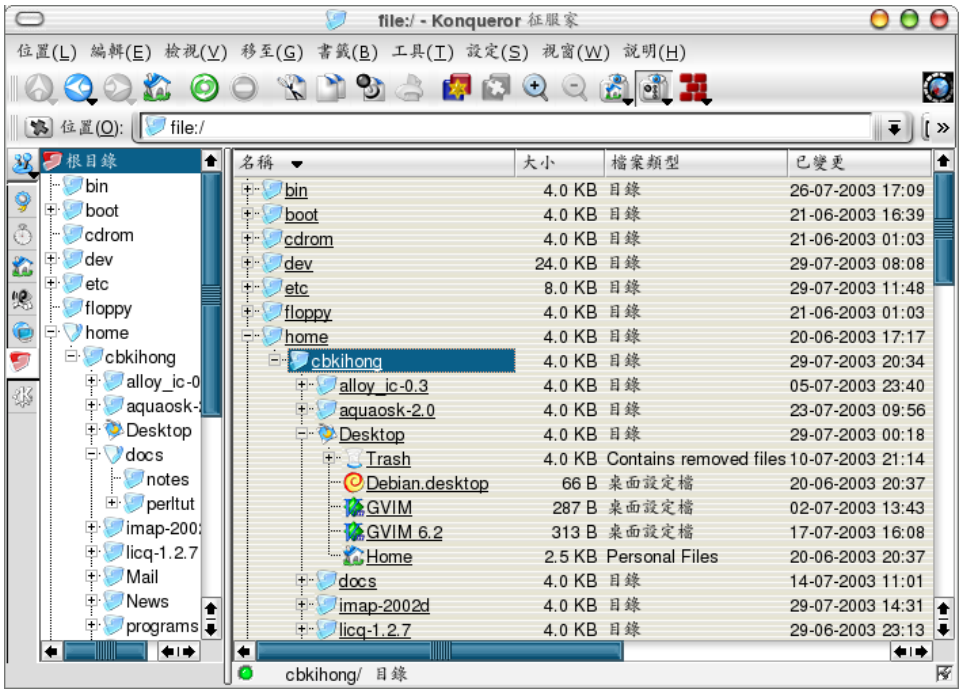

Figure D.1: Directory Structure In Tree Representation

indicator in the inode (see below). A few other file types are defined, yet they are specialized file types that are seldom used directly, so I am not going into details here.

D.2.2 Symbolic Links and Hard Links

The concept of links is foreign to users of many other operating systems, e.g. MS-DOS or Windows. Unix supports two types of links, namely **symbolic links** and **hard links**. Symbolic links and hard links in Unix are similar in sense to symbolic references and references in Perl, respectively. In many situations, symbolic links and hard links have similar effects. However, under the hood they are implemented in very different ways which give rise to different behaviours in certain situations. Because the concept of hard links is not very clearly explained in many Unix literature, I decided to

Directory	Purpose
`/bin`	Essential user programs, e.g. shells, file tools
`/boot`	Core system files for system bootup
`/dev`	Device files (remember devices are represented as files?)
`/etc`	System-wide configuration files for system and user programs
`/home`	User accounts: files private to individual users
`/mnt`	Mount point for foreign filesystems
`/root`	User account of system administrator (root)
`/sbin`	System administration commands used only by root
`/tmp`	Temporary files
`/usr`	User programs that are installed system-wide, and related files
`/var`	Data files written by system programs, e.g. email and various logs

Table D.1: Major Unix Standard Directories and Their Purposes

elaborate a little bit on this concept in this section.

Symbolic links are easier to understand. It works like "shortcuts" (the .lnk) files on Windows. Basically, a symbolic link is a special type of file that stores an alternative access path. When the symbolic link is accessed, the stored path is used for access. Symbolic links are frequently used to create shortcuts to very long paths, such that a shorter path is used instead. For software distribution sites, they are also used to maintain a static URL to the latest release. For example, on a certain system there are directories and files as follows:

```
/
  pub/
    download/
      myprog_current.zip -> /usr/share/devel/myprog/current.zip
  usr/
    share/
      devel/
        myprog/
          current.zip -> myprog_3.0.zip
          myprog_1.0.zip
          myprog_2.0.zip
          myprog_2.2.zip
          myprog_3.0.zip
```

The "`->`" indicates a symbolic link. On the left is the name of the symbolic link (the alias), and on the right is the alternative path to follow. Say `myprog` is a software developed. In this example, to access the current version of `myprog`, the path is `/pub/download/myprog_current.zip`. When the file is accessed through this symbolic link, the system would then try to access it at `/usr/share/devel/myprog/current.zip`, which in turn is also a symbolic link to `myprog_3.0.zip`, the archive of the latest release. Therefore, essentially, the path `/usr/share/devel/myprog/myprog_3.0.zip` is used to access the file. This example serves both purposes. It shortens the URL, and users can always get the latest release with the same URL, provided the symbolic links are properly maintained to point to the latest release. You may also wonder why I create two symbolic links instead of having `myprog_current.zip` pointing to `myprog_3.0.zip` directly. This is for convenience in management, so that no matter the user is in `/pub/download` or `/usr/share/devel/myprog` he or she can still easily locate the latest version, and notice that by doing so the symbolic link in `/pub/download` needs not be updated when a new release is placed in `/usr/share/devel/myprog`, only `current.zip` needs to be.

To create a symbolic link, use the `ln` command, with the `-s` switch. For example,

```
cbkihong@cbkihong:~/test$ cd pub/download
cbkihong@cbkihong:/pub/download$ ln -s /usr/share/devel/myprog/current.zip ¶
    myprog_current.zip
```

Hard links are more difficult to understand. Recall that in the previous section I described what a filesystem is. By creating a filesystem, indexing facilities that allow the operating system to quickly locate which disk blocks a certain file occupies are created on the disk. This serves a similar purpose as the map in a traveller's pocket. When a file is created on the disk, an **inode** (index node) is created for the file which contains attributes such as the owner and group (see next section) identifiers, the times of modification and access, the file type and locations of disk blocks containing the

file content (pointers). Finally, the operating system needs to add to the inode of the containing directory the pointer to the newly created inode so that the file is added to the directory entry. This pointer is actually a hard link. Hard links work at the level of inodes to allow direct access of the file being pointed to. Therefore, for every inode there is at least one hard link that points to it, from the containing directory file entry. If we create an additional hard link to a file, that means a new pointer to the file inode is added. Here is a sequence of commands which serves as an example:

```
cbkihong@cbkihong:~/test$ touch file1.txt
cbkihong@cbkihong:~/test$ ln file1.txt file2.txt
cbkihong@cbkihong:~/test$ touch file3.txt
cbkihong@cbkihong:~/test$ ls -li
 342533 -rw-r--r--    2 cbkihong cbkihong        0 2003-07-30 17:56 file1.txt
 342533 -rw-r--r--    2 cbkihong cbkihong        0 2003-07-30 17:56 file2.txt
 342531 -rw-r--r--    1 cbkihong cbkihong        0 2003-07-30 17:59 file3.txt
cbkihong@cbkihong:~/test$ echo "ABCD" > file1.txt
cbkihong@cbkihong:~/test$ ls -li
 342533 -rw-r--r--    2 cbkihong cbkihong        5 2003-07-30 18:01 file1.txt
 342533 -rw-r--r--    2 cbkihong cbkihong        5 2003-07-30 18:01 file2.txt
 342531 -rw-r--r--    1 cbkihong cbkihong        0 2003-07-30 17:59 file3.txt
```

The `touch` command creates an empty file. The `ln` command without any switches creates a hard link. Here, `file2.txt` is created as an additional hard link to the inode of `file1.txt`. The `ls` command displays a listing of files. Here, two switches `-l` and `-i` are given. If multiple switches are provided on the command line, you can combine them into `-li` (ordering does not matter). The `-l` option causes the long listing to be displayed, with the permission values, owner, group, date of the last modification and the file size. If the `-i` switch is given, the inode number that the file entry points to is inserted as the first column. We can see that `file1.txt` and `file2.txt` points to the same inode, while `file3.txt` points to a different inode. The third column is the number of hard links pointing to the inode. This number is stored at each inode. For regular files, this number is usually one, as explained previously. However, because we have manually created a new hard link as `file2.txt`, the number displayed is 2. Also notice that the two entries are exactly identical, except the name. Here, the text "ABCD" (with line terminating character) is written into `file1.txt`, and both entries are updated.

You may wonder why the filesystem has to keep track of the number of hard links pointing to an inode. If you have read the chapter on references in Perl you will find a close resemblance with the Perl garbage collection mechanism. That is, the inode would not be freed (deleted) until the number of hard links to it drops to 0. Consider the example again. If at this point we delete `file1.txt`, the number drops to 1, but because we still have a hard link that points to it from the entry `file2.txt`, the inode is not freed. It is not until when `file2.txt` is deleted, that the inode and other disk blocks that are associated with this file will eventually be freed.

```
cbkihong@cbkihong:~/test$ rm file1.txt
cbkihong@cbkihong:~/test$ ls -li
 342533 -rw-r--r--    1 cbkihong cbkihong        5 2003-07-30 18:01 file2.txt
 342531 -rw-r--r--    1 cbkihong cbkihong        0 2003-07-30 17:59 file3.txt
cbkihong@cbkihong:~/test$ rm file2.txt
cbkihong@cbkihong:~/test$ ls -li
 342531 -rw-r--r--    1 cbkihong cbkihong        0 2003-07-30 17:59 file3.txt
```

Occasionally you will find some files (and directories) with the "." name prefix, e.g. ".kde3/" and ".vimrc". These files (the "dot" files) are usually found in private user accounts in the `/home` tree, but in most cases you won't see them at all because the "dot" files are hidden by default. These files are usually created by user applications to store user-specific configuration and data files, because the user account is usually the only directory that applications written by users can write to that is private to the user. The `/etc` tree is owned by root and only system-wide configuration can be made there by the system administrator, and the `/tmp` tree are for temporary files only and need to be periodically cleaned up to avoid accumulation of useless files, thus data files that are to be kept cannot be placed into these directories. Due to the large number of applications installed in the system, very likely large numbers of these data files have to be created in the user account. To avoid clutter and prevent the user from accidentally deleting these files, Unix hides these files by default, unless you pass the `-a` option to the `ls` command.

Are you wondering why I suddenly jump from my hard links discussion to "dot" files? That is because I am going to introduce to you two special "dot" files that are present in every directory, namely "." and "..". MS-DOS also has the notion of these two special files. They are automatically created in a directory when the directory is created. They are in fact hard links to the current directory file, and the parent directory file respectively. Therefore, they appear as directories with the `ls -la` command. ".." is used to refer to the parent directory in the directory tree. For example, with the `cd` command, you can specify the name of a subdirectory to go into it, but when you need to return to the parent directory, you can use the command

```
cd ..
```

As another example, `cat ../../README.txt` outputs the file `README.txt` two levels upward. For example, if you are in the directory `/home/cbkihong/docs/perltut` when you type the above command, then the Unix shell will try to find the file at `/home/cbkihong/README.txt` and display its content. The "." directory points to the current directory. It seems to be not useful at all. However, I can find at least one use of it. For security reasons, many Unix installations would not put the current directory, that is "." into the environment variable `PATH`, especially for the root user. Therefore, if you have an executable file, say `myprog`, in your current directory by just typing `myprog` the program will not be started at all, because `PATH` is the executable search path that is used if the path to the executable is not specified. Directories not listed in this variable will not be searched for executables. In this case, you need to qualify it with the directory where it can be found, by

```
./myprog
```

If the target file is not an executable, the "./" prefix is generally optional and well understood if absent. Therefore, `cat docs/README` is generally understood to be `cat ./docs/README`. Because of these two special "dot" files, you will find that the number of hard links pointing to directories is never one. As a bare minimum there are two, one due to "." and the other one from the parent directory file inode. If there are subdirectories, then ".." in the subdirectories will add to the number of hard links pointing to the current directory inode. If there are extra hard links created manually to the current directory, then there are even more. For example,

```
/
  bin/
  etc/
  home/
    cbkihong/
```

The /home directory inode has 3 hard links pointing to it. However, there are 5 for /, that is because there is no parent directory for /, and for consistency the ".." in / also points to itself. Together with "." and ".." from the 3 subdirectories, the hard link count is therefore 5.

Symbolic links are more widely used because symbolic links only stores an alternative path name and can be used provided the alternative path is accessible from the directory tree. That is, symbolic links may point to a destination that is on a different filesystem. For example, a symbolic link on my Linux reiserfs partition may point to a file on the Windows FAT partition, which does not even have the notion of hard links (did I tell you that you cannot create a hard link on a mounted Windows partition at all?). On the other hand, hard links cannot cross filesystems, and they are only supported on Unix-compatible filesystems, so their areas of application are rather limited.

D.2.3 Permission and Ownership

Because Unix is from the ground up a multi-user operating system, a permission and ownership system has to be in place to control who have access to resources and how they can access them. In Unix filesystems, every file has an owner, the user who created the file on the filesystem. Apart from the owner, each file is associated with a group. The `ls` command with the `-l` switch causes the directory listing to be printed in the long format, with the name of the owner and group printed in the third and fourth column, respectively. The first column is the access permissions. In this column, the first character indicates the file type. Customarily, regular files are represented by a hyphen "-", directories by "d" and symbolic links by "l".

```
cbkihong@cbkihong:~/docs/perltut$ ls -l
drwxr-xr-x    2 cbkihong users         477 2003-06-20 14:25 images
-rw-r--r--    1 cbkihong users      657800 2003-06-20 14:28 perltut.pdf
-rw-r--r--    1 cbkihong users     4012769 2003-06-20 14:28 perltut.ps
-rw-r--r--    1 cbkihong users        3887 2003-06-19 19:17 perltut.tex
```

In this example, the files have the owner "cbkihong" and belong to the group "users". To control how different users can access the files, three sets of permission bits are assigned to each file which specify permissions that are given to the owner, group members and everybody else. The permission values are reflected by the first column of the file listing obtained by the `ls` command above. The first character indicate the type of the file. The remaining nine characters represent the permission values. Here is a summary of what each character means with respect to a directory and a regular file:

Let's take the first two entries in the example listing above as an example. For the "images" directory, we divide the permission values into three sets:

```
file type  owner  group   everybody else
    d       rwx    r-x         r-x
```

Everybody on the system can read the directory listing and enter the directory. However, only the owner "cbkihong" can add new files or remove files from the directory. To read the directory listing means you get a list of names that represent the files (including subdirectories and symbolic links etc.) in the directory. For example, if you enable the "x" bit but not the "r" bit, the command `ls images/` will fail because this operation involves getting the directory listing. However, the command `cat images/README.txt` will be successful if the file `README.txt` exists in the "images"

Regular File

Bit	Value	Meaning
r	4	Read file content
w	2	Modify the file content
x	1	Execute the file

Directory

Bit	Value	Meaning
r	4	Read directory listing
w	2	Create/delete files in the directory
x	1	Enter the directory

Table D.2: Filesystem Permission values

directory because directory listing is not involved in the operation. On the other hand, if the "r" bit is enabled but not the "x" bit, you can see the list of files in the directory, but you cannot access them. Changing into the directory with the command `cd` will also fail. Because such a combination of permission bits is somewhat nonsense in practical use, for directories usually the "r" and "x" bits go together — either you enable or disable both of them, but not enabling one and not the other.

Unix groups are not frequently used in practice, but they can be good for sharing of files among users on the system in a simple way. For example, a file server in a company may have a "managers" group whose members consist of managers from all departments. The server may have a directory called "reports" containing reports prepared by the managerial for all staff. If the directory is not in the `/home` tree, it is very likely owned by the system administrator (root). However, the administrator may set the group to "managers" and set the group "w" bit to allow the managers to put their reports into the directory, while other staff users only have read access. A possible configuration is shown below:

```
drwxrwxr-x    2 root  managers         477 2003-07-20 14:38 reports
```

For the "perltut.pdf" file in the sample file listing above, the permission values are as follows:

```
file type  owner  group   everybody else
    -       rw-    r--          r--
```

That means everybody can read the file, but only the owner can modify it. Some people have misconceptions on Unix permissions that one needs directory write permission to modify a file in the directory. The fact is only the file write permission is needed. Also, some may think to delete a file one needs writable permission to that file. Only the directory write permission is needed in this case. If you understand my description above fully, you are not going to make these kinds of mistakes.

Note that internally (for example, in the inodes) owners and groups are represented by numbers instead of names such as "cbkihong" and "users". That is because storage of integers uses less space compared with names. On most systems, the `/etc/passwd` stores the mapping between user ID and username, while `/etc/group` stores the mapping between group ID and group name. They are used to resolve the mnemonic names for display as output of commands such as `ps` and `ls` etc.

In Table D.2 you see a column with the heading "Value". Internally, to store the permission values in a more compact form the permissions are encoded as a number. Take the "images" directory as an example. The permission value "drwxr-xr-x" may be converted to numeric representation as follows:

Owner (rwx): $4+2+1=7$
Group (r-x): $4+1=5$
Everybody else (r-x): $4+1=5$

As already described, the character "d" is only an indication that this is a directory, and it's not a permission value. Therefore, the numeric representation is 755, by concatenating the permission values for the owner, group and everybody else. Please note that permission values of symbolic links are not used in practice, and are set to 777 (lrwxrwxrwx) by default.

D.2.4 Processes

Once executable permission is applied to a file, it can then be executed by the system. A program in execution is called a process. Each process has an owner and is associated with a group, similar to the case of files on a filesystem. Each process is associated with four identifiers. Apart from the user ID and group ID, a process also has an **effective user ID** and an **effective group ID**. In general, when an executable file is being executed, the user ID is that of the user who executed the program, and the group ID is that assigned to the user when the user account was created. In general, the effective user ID is the same as user ID, and the effective group ID is the same as group ID. They are different only if the file being executed has either the setuid or setgid bit set, which are two additional permission bits that are useful only to executable files and will be described in the next section. They are seldom needed, and their use are usually not justified unless with good reasons.

The enforcement of an ownership system on processes prevents unauthorized users from modifying the state of the processes, for example, to terminate them. In general, only root or the users whose user ID matches the user ID or effective user ID of a process are allowed to change the state of a process. You can change the state of a process by sending it a **signal**, that is, a message sent from the operating system kernel to a process. From a user's point of view, a process can be in one of several states: running, suspended or terminated. We can use the **kill** command to send a signal to a process. The signals that are most frequently used include `SIGHUP` (1), `SIGINT` (2), `SIGKILL` (9), `SIGSTOP` (19) and `SIGCONT` (18). The way `SIGHUP` is handled is process-specific. It is generally widely supported that when a daemon process receives this signal, it rereads its configuration files. This is convenient for system administrators to effectuate changes made to the configuration files without restarting the daemon process. The `SIGINT` signal is what is sent to a process when the user presses Ctrl-C. In most cases, the process is terminated. However, some processes are defined to catch the signal and thus prevent it from being terminated. For example, Ctrl-C is a combination key defined in emacs, so it has to catch the signal. In this case, you may try to send the `SIGTERM` (15) signal. Runaway processes can generally be abruptly terminated by the `SIGKILL` signal. The `SIGSTOP` signal causes the process to be suspended. Both `SIGKILL` and `SIGSTOP` cannot be caught by any processes, and therefore provided you have the permission to change the state of the process these two signals should succeed, at least in theory. However, there are a few occasions when the process, or even other parts of the operating system are ignoring these signals. This is a sign of inauspiciousness and you are advised to restart your system if this happens to you (but probably by then it is already too late). For a suspended process, you can put it back to running state by sending it a `SIGCONT` signal. You can send a signal using `kill` in several forms, as shown in the following examples:

```
kill -SIGHUP 826
kill -HUP 826
kill -1 826
```

You can use the name of the signal or the signal ID to specify the signal to send to the process. Because all Unix signal names actually start with the prefix "SIG", you may omit this prefix to reduce the amount of typing. The last argument to `kill` is the process ID. Each process has an ID to uniquely identify a process. You should check the process ID by using the `ps` command, which is displayed on the far left. Most systems also have a `killall` command which frees you from the need of looking up the process ID. For example,

```
killall -SIGINT myprog
```

which tries to terminate all instances of the program myprog. However, `kill` is a more reliable choice on some Unix variants, or where `killall` is not available.

A side note about the executable and read permission of executable files. I have recently seen a question raised on a Unix forum about making a shell script executable but not readable by other users on the system. The answer is that is not possible to use the permission 711 (rwx–x–x). There are two main types of executable programs. Either they are compiled into an executable object code (binary) format that can be executed directly (e.g., compiled programs written in C), or shell scripts (e.g. perl or sh scripts). If the executable is in an object format supported by the operating system kernel, it can be directly loaded and executed by the kernel. Today two main binary formats are supported on Unix, namely **a.out** and **ELF** (Executable and Linkable Format). a.out is well supported, but is quickly replaced by ELF on many Unix platforms. In the case of executable scripts, on the other hand, the kernel needs to load the interpreter instead, and the interpreter executes the script instead. To execute the script, the interpreter needs to have read access of the script. That's why you can't use 711 as the script permission if everybody needs to execute it. This works for compiled programs, though.

D.2.5 The Special Permission Bits

Apart from the 9 permission bits that can be set for every file, there are three special option bits that may be applied, two of which pertain to executable files. These bits are known as **setuid**, **setgid** and **sticky** respectively. They may be set using the `chmod` command.

If you are granted access to a Unix shell account, possibly one of the first commands you will be asked to use is `passwd`. This command lets you change your password. On some Unix platforms, some other functionalities are also available, for example, to change the login shell. Traditionally on Unix, user information are stored in the file `/etc/passwd`, containing the username, user ID, home directory, login shell and an encrypted password of users. As encrypted passwords in traditional Unixes are well known to be easy to break, today many systems are configured to use **shadowed passwords**, whose encrypted (strictly speaking, hashed by a cryptographically strong algorithm such as MD5) passwords are stored in a separate file `/etc/shadow` which is only readable by root, in contrast to `/etc/passwd` which is readable by everybody. The `passwd` file has to be readable by everybody because its file content is read by a number of standard Unix programs, such as `finger`, which can be run by all users on the system to display information about the user (it may also be executed remotely, but it's usually disabled to minimize security risks). The `passwd` file is owned by root. Have you wondered how the `passwd` program which is executed by a regular user can modify this file which is only writable by root? The fact is, the `passwd` is one of the few programs in

a standard Unix installation that has the setuid bit set, which allows it to take on the privilege of the root user.

When an executable with the setuid bit set is being executed, the effective user is set to the owner of executable instead of the user who executed the program. Similar is the case for the setgid bit. Therefore, essentially, the setuid and setgid bits are used to escalate the privileges of the executable. It is also for this reason that setuid and setgid bits should not be set unless with strong reasons, and programs with these bits on should be examined closely to ensure they do not have any inherent vulnerabilities that may be taken advantage by attackers. For Perl programs, for example, the program `suidperl` which is itself a setuid version of the perl interpreter is actually used to execute a Perl program with the setuid or setgid bit set. As mentioned in Chapter 10, runtime taint check is performed automatically. An executable may be set setuid or setgid using the `chmod` command:

```
chmod u+s executable    # setuid
chmod g+s executable    # setgid
```

For example, `passwd`, a setuid program shows up as this:

```
-rwsr-xr-x    1 root     root        24376 2003-09-14 05:53 /usr/bin/passwd
```

The sticky bit serves a totally different purpose. Many programs in execution need to create large number of temporary files. In order to avoid cluttering the home directory of the respective users, they are customarily created under the `/tmp` tree. A check of the `/tmp` directory shows that it is owned by root, and the group is root. Here is the entry displayed by `ls -l /`:

```
drwxrwxrwt    8 root     root         4096 2003-09-22 14:47 tmp
```

Because the directory is writable by everybody (the w bit is enabled for others), programs executed by regular users can use it as a swap space to create temporary files. However, this also creates a problem. Note that while any regular user may create anything in this directory, they may as well remove anything in this directory including files and directories created by another user. A malicious user may keep on removing other users' files to inconvenience them. The workaround is a sticky bit applied on the `/tmp` directory. If the sticky bit is enabled, the system will not allow you to modify any entries in the directory unless you are the owner specified at the entry, or the root user. You may enable sticky mode by the command shown below:

```
chmod +t somedir/
```

t stands for a sticky bit. You can see from above that it appears instead of x in the listing of `/tmp`. Here is an example of some entries in `/tmp` on my Debian Linux system:

```
cbkihong@cbkihong:/$ ls -l /tmp
-rw-r--r--    1 cbkihong cbkihong        0 2003-09-22 14:25 AcroEiBBIX
drwx------    2 cbkihong cbkihong     4096 2003-09-22 14:23 kde-cbkihong
```

Unless you are the owner, that is "cbkihong", you cannot rename or remove these two entries.

Appendix E

BNF Grammar of Selected Functions

Here, I will present the full grammar of selected functions using the Backus-Naur Form (BNF), a widely-used and systematic method of representing syntax (grammar) of computer languages.

E.1 `sprintf()`/`printf()`

```
<placeholder> ::= % <more_attr> <conv_type>
<conv_type> ::= % | c | s | d | u | o | x | X | e | E | f | g | G | b | p | n | ε
<more_attr> ::= <param_idx> <flags> <vector> <min_width> <max_width> <size> <
    format_idx>
<param_idx> ::= <num> | ε
<flags> ::= <flag_base> <flag_signprefix> <flag_padding>
<flag_base> ::= # | ε
<flag_signprefix> ::= [ ␣ | + ] | ε
<flag_padding> ::= [ 0 | - ] | ε
<min_width> ::= <num> | <arg> | ε
<max_width> ::= . [ <num> | * ] | ε
<arg> ::= * [ <num> $ | ε ]
<size> ::= l | h | q | L | ll | ε
```

Appendix F

In The Next Edition

The following topics or modifications are planned for inclusion in the next edition of this Perl 5 Tutorial. If you have any other suggestions on other topics of interest or amendments to the content of this edition, please feel free to use my feedback forum or my email feedback form to let me know. The Web links can be found on page 2.

- ⋆ more full code examples and illustrative figures
- ⋆ more advanced CGI scripting techniques and more on `CGI`
- ⋆ threading and `fork()`ing new processes
- ⋆ mod_perl
- ⋆ Database access with `DBI` and `DBD::*`
- ⋆ introduction to XS
- ⋆ internationalization support in Perl and PerlIO layers
- ⋆ GUI building with Tk
- ⋆ more examples on complex data structures and algorithms implementations
- ⋆ socket programming and existing CPAN classes (e.g. `LWP`, `Net::*`)
- ⋆ "heredoc" quotation syntax (I'm not fond of it but some people do use it)
- ⋆ Regular expressions (Perl extensions)
- ⋆ `Exporter` module in object-oriented programming
- ⋆ more object-oriented design principles and examples
- ⋆ coverage of `File::*` and `IO::*` classes
- ⋆ operator overloading
- ⋆ Command line parameter parsing with `GetOpt::Long`
- ⋆ ties and object persistence

Index

Preamble

The intent of this document is to state the conditions under which this package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

Definitions

- ★ "Package" refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.
- ★ "Standard Version" refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder.
- ★ "Modified Version" refers to such a Package that has been modified but does not qualify as a "Standard Version".
- ★ "Copyright Holder" is whoever is named in the copyright or copyrights for the package.
- ★ "You" is you, if you're thinking about copying or distributing this Package.
- ★ "Reasonable copying fee" is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)
- ★ "Freely Available" means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

Terms and Conditions

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.

2. In any case, the terms and conditions set forth in this document should not be modified without specific prior written permission of the Copyright Holder.

3. You may apply bug fixes, portability fixes and other modifications derived from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version. You may distribute the Standard Version of this Package without restriction.

4. Modifications not meeting the criteria aforementioned shall be considered a Modified Version. You are required to insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:

    a) place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as ftp.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.

    b) use the modified Package only within your corporation or organization.

    c) make other distribution arrangements with the Copyright Holder.

5. You may freely distribute Modified Versions of this Package provided that you do at least ONE of the following:

    a) provide instructions on where to get the Standard Version of this Package.

    b) make other distribution arrangements with the Copyright Holder.

6. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own.

7. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The End