Programming Perl

Kirrily Robert
Paul Fenwick
Jacinta Richardson

Programming Perl
by Kirrily Robert, Paul Fenwick, and Jacinta Richardson

Open Publications License 1.0

This document is a revised and edited copy of the Introduction to Perl and Intermediate Perl training notes originally created by Kirrily Robert and Netizen Pty Ltd. These revisions were made by Paul Fenwick and Jacinta Richardson.
Copies of the original training manuals can be found at http://sourceforge.net/projects/spork
This training manual is maintained by Perl Training Australia, and can be found at http://www.perltraining.com.au/notes.html
This is version 1.0 of Perl Training Australia's "Programming Perl" training manual.

Table of Contents

List of Tables

Chapter 1. About Perl Training Australia

Training

Perl Training Australia (http://www.perltraining.com.au) offers quality training in all aspects of the Perl programming language. We operate throughout Australia and the Asia-Pacific region. Our trainers are active Perl developers who take a personal interest in Perl's growth and improvement. Our trainers can regularly be found frequenting online communities such as Perl Monks (http://www.perlmonks.org/) and answering questions and providing feedback for Perl users of all experience levels.

Our primary trainer, Paul Fenwick, is a leading Perl expert in Australia and believes in making Perl a fun language to learn and use. Paul Fenwick has been working with Perl for over 10 years, and is an active developer who has written articles for *The Perl Journal* and other publications.

Doctor Damian Conway, who provides many of our advanced courses, is one of the three core Perl 6 language designers, and is one of the leading Perl experts worldwide. Damian was the winner of the 1998, 1999, and 2000 Larry Wall Awards for Best Practical Utility. He is a member of the technical committee for OSCON, a columnist for The Perl Journal, and author of the book "Object Oriented Perl".

Consulting

In addition to our training courses, Perl Training Australia also offers a variety of consulting services. We cover all stages of the software development life cycle, from requirements analysis to testing and maintenance.

Our expert consultants are both flexible and reliable, and are available to help meet your needs, however large or small. Our expertise ranges beyond that of just Perl, and includes Unix system administration, security auditing, database design, and of course software development.

Contact us

If you have any project development needs or wish to learn to use Perl to take advantage of its quick development time, fast performance and amazing versatility; don't hesitate to contact us.

Table 1-1. Perl Training Australia's contact details

Phone:	03 9354 6001
Fax:	03 9354 2681
Email:	contact@perltraining.com.au
Webpage:	http://www.perltraining.com.au/
Address:	104 Elizabeth Street, Coburg VIC, 3058

Chapter 2. Introduction

Welcome to Perl Training Australia's *Programming Perl* training course. This is a four-day module in which you will learn how to program the Perl programming language.

Credits

This course is based upon the Introduction to Perl and Intermediate Perl training modules written by Kirrily Robert of Netizen Pty Ltd.

Course outline

Day 1

- What is Perl?
- Introduction to perldoc
- Creating and running a Perl program
- Variable types
- Operators and Functions

Day 2

- Conditional constructs
- Subroutines
- Regular expressions
- Practical exercises

Day 3

- Revise assumed knowledge
- References and complex data structures
- Introduction to modules and packages
- Writing packages and modules
- Using Perl objects

Day 4

- Advanced regular expressions
- File I/O
- System interaction
- Bonus material

Assumed knowledge

This training module assumes the following prior knowledge and skills:

- You have programmed in least one other language and you:
  - Understand the concept of variables, including arrays and pointers/references
  - Understand conditional and looping constructs
  - Understand the use of user defined functions

Platform and version details

Perl is a cross-platform computer language which runs successfully on approximately 30 different operating systems. However, as each operating system is different this does occasionally impact on the code you write. Most of what you will learn will work equally well on all operating systems; your instructor will inform you throughout the course of any areas which differ.

At the time of writing, the most recent stable release of Perl is 5.8.6, however older versions of Perl (particularly 5.6.1 and 5.005) are still common. Your instructor will inform you of any features which may not exist in earlier versions.

The course notes

These course notes contain material which will guide you through the topics listed above, as well as appendices containing other useful information.

The following typographical conventions are used in these notes:

System commands appear in **this typeface**

Literal text which you should type in to the command line or editor appears as `monospaced font`.

Keystrokes which you should type appear like this: **ENTER**. Combinations of keys appear like this: **CTRL**-**D**

```
Program listings and other literal listings of what appears on the
screen appear in a monospaced font like this.
```

Parts of commands or other literal text which should be replaced by your own specific values appears *like this*


Notes and tips appear offset from the text like this.


Notes which are marked "Advanced" are for those who are racing ahead or who already have some knowledge of the topic at hand. The information contained in these notes is not essential to your understanding of the topic, but may be of interest to those who want to extend their knowledge.


Notes marked with "Readme" are pointers to more information which can be found in your textbook or in online documentation such as manual pages or websites.


Notes marked "Caution" contain details of unexpected behaviour or traps for the unwary.

Other materials

In addition to these notes, it is highly recommend that you obtain a copy of Programming Perl (2nd or 3rd edition) by Larry Wall, et al., more commonly referred to as "the Camel book". While these notes have been developed to be useful in their own right, the Camel book covers an extensive range of topics not covered in this course, and discusses the concepts covered in these notes in much more detail. The Camel Book is considered to be the definitive reference book for the Perl programming language.

The page references in these notes refer to the *3rd edition* of the Camel book, unless otherwise stated. References to the 2nd edition will be shown in parentheses.

Chapter 3. What is Perl

In this chapter...

This section describes Perl and its uses. You will learn about this history of Perl, the main areas in which it is commonly used, and a little about the Perl community and philosophy. Lastly, you will find out how to get Perl and what software comes as part of the Perl distribution.

Perl’s name and history

Perl was originally written by Larry Wall as a tool to assist him with a re-write of the then popular "rn" news-reader. Larry found himself desiring a language which tied together the best features of diverse languages such as C, shell, awk and sed, and wrote Perl to fill this need. Perl was a huge success with system administrators, and so development of the language flourished. Due to Perl’s popularity, Larry never finished the rewrite of rn.

Perl allegedly stands for "Practical Extraction and Reporting Language", although some people swear it stands for "Pathologically Eclectic Rubbish Lister". In fact, Perl is not an acronym; it’s a shortened version of the program’s original name, "Pearl". According to Larry Wall, the name was shortened because all other good Unix commands were four letters long, so shortening Perl’s name would make it more popular.

When we talk about the language it’s spelled with a capital "P" and lowercase "erl", not all capitals as is sometimes seen (especially in job advertisements posted by contract agencies). When you’re talking about the Perl interpreter, it’s spelled in all lower case: **perl**.

Perl has been described as everything from "line noise" to "the Swiss-army chain-saw of programming languages". The latter of these nicknames gives some idea of how programmers see Perl - as a very powerful tool that does just about everything.

Typical uses of Perl

Text processing

Perl’s original main use was text processing. It is exceedingly powerful in this regard, and can be used to manipulate textual data, reports, email, news articles, log files, or just about any kind of text, with great ease.

System administration tasks

System administration is made easy with Perl. It’s particularly useful for tying together lots of smaller scripts, working with file systems, networking, and so on.

CGI and web programming

Since HTML is just text with built-in formatting, Perl can be used to process and generate HTML. For many years Perl was the de facto language for web development, and is still very heavily used today. There are many freely available tools and scripts to assist with web development in Perl.

Database interaction

Perl's DBI module makes interacting with all kinds of databases --- from Oracle down to comma-separated variable files --- easy and portable. Perl is increasingly being used to write large database applications, especially those which provide a database backend to a website.

Other Internet programming

Perl modules are available for just about every kind of Internet programming, from Mail and News clients, interfaces to IRC and ICQ, right down to lower level socket programming.

Less typical uses of Perl

Perl is used in some unusual places as well. The Human Genome Project relies on Perl for DNA sequencing, NASA use Perl for satellite control, PDL (Perl Data Language, pronounced "piddle") makes number-crunching easy, and there is even a Perl Object Environment (POE) which is used for event-driven state machines.

What is Perl like?

The following (somewhat paraphrased) article, entitled "What is Perl", comes from The Perl Journal (http://www.tpj.com/) (Used with permission.)

> *Perl is a general purpose programming language* developed in 1987 by Larry Wall. It has become the language of choice for WWW development, text processing, Internet services, mail filtering, graphical programming, and every other task requiring portable and easily-developed solutions.
>
> *Perl is interpreted.* This means that as soon as you write your program, you can run it -- there's no mandatory compilation phase. The same Perl program can run on Unix, Windows, NT, MacOS, DOS, OS/2, VMS and the Amiga.
>
> *Perl is collaborative.* The CPAN software archive contains free utilities written by the Perl community, so you save time.
>
> *Perl is free.* Unlike most other languages, Perl is not proprietary. The source code and compiler are free, and will always be free.
>
> *Perl is fast.* The Perl interpreter is written in C, and more than a decade of optimisations have resulted in a fast executable.
>
> *Perl is complete.* The best support for regular expressions in any language, internal support for hash tables, a built-in debugger, facilities for report generation, networking functions, utilities for CGI scripts, database interfaces, arbitrary-precision arithmetic --- are all bundled with Perl.
>
> *Perl is secure.* Perl can perform "taint checking" to prevent security breaches. You can also run a program in a "safe" compartment to avoid the risks inherent in executing unknown code.

Perl is open for business. Thousands of corporations rely on Perl for their information processing needs.

Perl is simple to learn. Perl makes easy things easy and hard things possible. Perl handles tedious tasks for you, such as memory allocation and garbage collection.

Perl is concise. Many programs that would take hundreds or thousands of lines in other programming languages can be expressed in a page of Perl.

Perl is object oriented. Inheritance, polymorphism, and encapsulation are all provided by Perl's object oriented capabilities.

Perl is flexible. The Perl motto is "there's more than one way to do it." The language doesn't force a particular style of programming on you. Write what comes naturally.

Perl is fun. Programming is meant to be fun, not only in the satisfaction of seeing our well-tuned programs do our bidding, but in the literary act of creative writing that yields those programs. With Perl, the journey is as enjoyable as the destination.

The Perl Philosophy

There's more than one way to do it

The Perl motto is "there's more than one way to do it" --- often abbreviated TMTOWTDI. What this means is that for any problem, there will be multiple ways to approach it using Perl. Some will be quicker, more elegant, or more readable than others, but that doesn't make the other solutions *wrong*.

A correct Perl program...

"... is one that does the job before your boss fires you." That's in the preface to the Camel book, which is highly recommended reading.

Perl makes it easy to perform many tasks, and was built with programmer convenience in mind. It is possible to develop Perl programs very quickly, although the resulting code is not always beautiful. This course aims to teach not only the Perl language, but also good programming practice in Perl as well.

Three virtues of a programmer

The Camel book contains the following entries in its glossary:

Laziness

The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labour-saving programs that other people will find useful, and document what you wrote so you don't have to answer so many questions about it. Hence, the first great virtue of a programmer.

Impatience

The anger you feel when the computer is being lazy. This makes you write programs that don't just react to your needs, but actually anticipate them. Or at least pretend to. Hence, the second great virtue of a programmer.

Hubris

Excessive pride, the sort of thing Zeus zaps you for. Also the quality that makes you write (and maintain) programs that other people won't want to say bad things about. Hence, the third great virtue of a programmer.

Three more virtues

In his "State of the Onion" keynote speech at The Perl Conference 2.0 in 1998, Larry Wall described another three virtues, which are the virtues of a community of programmers. These are:

- Diligence
- Patience
- Humility

You may notice that these are the opposites of the first three virtues. However, they are equally necessary for Perl programmers who wish to work together, whether on a software project for their company or on an Open Source project with many contributors around the world.

Share and enjoy!

Perl is Open Source software, and most of the modules and extensions for Perl are also released under Open Source licenses of various kinds (Perl itself is released under dual licenses, the GNU General Public License and the Artistic License, copies of which are distributed with the software).

The culture of Perl is fairly open and sharing, and thousands of volunteers worldwide have contributed to the current wealth of software and knowledge available to us. If you have time, you should try and give back some of what you've received from the Perl community. Contribute a module to CPAN, help a new Perl programmer to debug her programs, or write about Perl and how it's helped you. Even buying books written by the Perl gurus (like many of the O'Reilly Perl books), or subscribing to publications such as The Perl Journal helps give them the financial means to keep supporting Perl.

Parts of Perl

The Perl interpreter

The main part of Perl is the interpreter. The interpreter is available for Unix, Windows, and many other platforms. The current version of Perl is 5.8.6, which is available from the Perl website (http://www.perl.com/) or any of a number of mirror sites (the Windows version is available from ActiveState (http://www.activestate.com/)).

Perl 6, a serious revision of the language, is under active development. Perl 6 will share many features in common with Perl 5, but will also provide a great many improvements and features.

Manuals/Documentation

Along with the interpreter come the manuals for Perl. These are accessed via the **perldoc** command or, on Unix systems, also via the **man** command. More than 30 manual pages come with the current version of Perl. These can be found by typing **man perl** (or **perldoc perl** on non-Unix systems). The Perl FAQs (Frequently Asked Questions files) are available in perldoc format, and can be accessed by typing **perldoc perlfaq**.

Perl Modules

Perl also comes with a collection of modules. These are Perl libraries which carry out certain common tasks, and can be included as common libraries in any Perl script. Less commonly used modules aren't included with the distribution, but can be downloaded from CPAN (http://www.cpan.org) and installed separately.

Chapter summary

- Common uses of Perl include
  - text processing
  - system administration
  - CGI and web programming
  - other Internet programming

- Perl is a general purpose programming language, distributed for free via the Perl website (http://www.perl.com/) and mirror sites.
- Perl includes excellent support for regular expressions, object oriented programming, and other features.
- Perl allows a great degree of programmer flexibility - "There's more than one way to do it".
- The three virtues of a programmer are Laziness, Impatience and Hubris. Perl will help you foster these virtues.
- The three virtues of a programmer in a group environment are Diligence, Patience, and Humility.
- Perl is a collaborative language - everyone is free to contribute to the Perl software and the Perl community.
- Parts of Perl include: the Perl interpreter, documentation in several formats and library modules.

Chapter 4. A brief guide to perldoc

Depending upon your operating system, the way in which you access Perl's on-line documentation may differ, but the information that is available should be the same on all systems.

This chapter discusses Perl's on-line help on Unix flavoured operating systems. On such systems, most of Perl's help files are also available as man pages. However, **man** is not always good at finding documentation embedded inside modules and programs, whereas **perldoc** is very good at it.

Using perldoc

Table 4-1. Getting around in perldoc

Action	Keystroke
Page down	**SPACE**
Page up	**b**
Quit	**q**

Exercise

On the command line, type **perldoc perl**. You will find yourself in the Perl documentation pages.

Language features and tutorials

Perl comes with a large amount of documentation detailing the language, as well as some tutorials to help you learn. Learning the entire language from these help files is not easy (that's why you have these notes), but they're a very useful reference material.

perldoc perl will provide you with a long list of help topics, and **perldoc perltoc** will provide you with the same list but with subsections, so you can easily search for what you're after.

You might notice that all the help files start with `perl`, such as `perlfunc` or `perlfaq`. This is so that the Unix man pages can have the same names as the perldoc pages. Try **man perlfunc** and you'll get the same information as **perldoc perlfunc**.

Feel free to experiment and read any pages that interest you. If you're working on an unfamiliar machine, you might find **perldoc perllocal** handy to see which extra modules have been installed. **perldoc perlmodlib** lists Perl's standard modules.

Looking up functions

If you're like most people, you'll occasionally forget the calling syntax or exact details of a particular function. Rather than having to flick through a weighty book, or read through all of **perldoc perlfunc**, there is an easier way to obtain the information that you're after. **perldoc -f *`function`*** lists all the information available about the desired function. Try

- perldoc -f split
- perldoc -f grep
- perldoc -f map

This is probably the most common use of **perldoc**.

Looking up modules

While using and writing modules are not covered in this course, as your experience with Perl grows you will find yourself dealing with modules more often. You can find information about any installed module simply by using **perldoc** ***module***. For example, **perldoc CGI** would tell you more about Perl's `CGI` module, which is very useful in developing interactive web-sites.

This also works for pragmas, of which we'll cover a few today. Try **perldoc strict** for more information on the `strict` pragma.

Chapter 5. Creating and running a Perl program

In this chapter...

In this chapter we will be creating a very simple "Hello, world" program in Perl and exploring some of the basic syntax of the Perl programming language.

Logging into your account

However you're doing this course, you will have access to a machine on which to perform the practical exercises. Your instructor will tell you the options available to you.

You should find that you have an `exercises/` directory available in your account or on your desktop. This directory contains example scripts and answers that are referred to throughout these notes.

Our first Perl program

We're about to create our first, simple Perl script: a "hello world" program. There are a couple of things you should know in advance:

- Perl programs (or scripts --- the words are interchangeable) consist of a series of statements
- When you run the program, each statement is executed in turn, from the top of your script to the bottom. (There are two special cases where this doesn't occur, one of which --- subroutine declarations --- we'll be looking at tomorrow)
- Each statement ends in a semi-colon
- Statements can flow over several lines
- Whitespace (spaces, tabs and newlines) is ignored in most places in a Perl script.

Now, just for practice, open a file called `hello.pl` in your editor. Type in the following one-line Perl program:

```
print "Hello, world!\n";
```

This one-line program calls the `print` function with a single parameter, the *string literal* `"Hello, world!"` followed by a newline character.

Save it and exit.

Incidentally, Appendix G contains a guide to pronouncing ASCII characters, especially punctuation. Perl makes use of many punctuation symbols, so this will help you translate Perl into spoken language, for ease of communication with other programmers.

Running a Perl program from the command line

We can run the program from the command line by typing in:

```
% perl hello.pl
```

You should see this output:

```
Hello, world!
```

This program should, of course, be entirely self-explanatory. The only thing you really need to note is the `\n` ("backslash N") which denotes a new line. If you are familiar with the C programming language, you'll be pleased to know that Perl uses the same notation to represent characters such as newlines, tabs, and bells as does C.

Executing code

Writing **perl** in front of all of our programs to execute them can be a bit of a pain. What if we want to be able to run our program from the command line, without having to always type that in?

Well... it depends on the operating system.

Various operating systems have different ways of determining how to react to different files. For example, Microsoft Windows uses file extensions while the various Unixes are completely indifferent to all parts of the filename. Some operating systems use properties you can set individually.

This can lead to some confusion when trying to write code to be cross-platform. Where Microsoft Windows will recognise that all files with a `.pl` extension should be passed to the Perl interpreter, how can we ensure that we've done everything for the other platforms as well?

The "shebang" line for Unix

Unix and Unix-like operating systems do not automatically realise that a Perl script (which is just a text file after all) should be executable. As a result, we have ask the operating system to change the file's permissions to allow execution:

```
% chmod +x hello.pl
```

Once the file is executable we also need to tell Unix how to execute the program. This allows the operating system to have many executable programs written in different scripting languages.

We tell the operating system to use the Perl interpretor by adding a "shebang" line (called such because the `#` is a "hash" sign, and the `!` is referred to as a "bang", hence "hashbang" or "shebang").

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

Of course, if the Perl interpreter were somewhere else on our system, we'd have to change the shebang line to reflect that.

This allows us to run our scripts just by typing:

```
% ./hello.pl
```

For security purposes, many Unix and Unix-like systems do not include your current directory in those which are searched for commands, by default. This means that if you try to invoke your script by typing:

```
% hello.pl                    # this doesn't work
```

you'll get the error: `bash: hello.pl: command not found`. This is why we prepend our command with the current working directory (**./hello.pl**).

The "shebang" line for non-Unixes

It's always considered a good idea for *all* Perl programs to contain a shebang line. This is helpful because it allows us to include command line options, which we'll cover shortly.

If your program will only ever be run on your single operating system then you can use the line:

```
#!perl
```

However it is considered good practice to use the traditional:

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

as this assists with cross-platform portability.

Comments

Comments in Perl start with a hash sign (#), either on a line on their own or after a statement. Anything after a hash is a comment up to the end of the line.

```
#!/usr/bin/perl
# This is a hello world program
print "Hello, world!\n";        # print the message
```

Command line options

Perl has a number of command line options, which you can specify on the command line by typing **perl *`options`* hello.pl** or which you can include in the shebang line. Let's say you want to use the `-w` command line option to turn on warnings:

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

(It's always a good idea to turn on warnings while you're developing something, and often once your code has gone into production, too.)

A full explanation of command line options can be found in the Camel book on pages 486 to 505 (330 to 337, 2nd Ed) or by typing **perldoc perlrun**.

Chapter summary

Here’s what you know about Perl’s operation and syntax so far:

- Perl programs typically start with a "shebang" line.
- statements (generally) end in semicolons.
- statements may span multiple lines; it’s only the semicolon that ends a statement.
- comments are indicated by a hash (#) sign. Anything after a hash sign on a line is a comment.
- `\n` is used to indicate a new line.
- whitespace is ignored almost everywhere.
- command line arguments to Perl can be indicated on the shebang line.
- the `-w` command line argument turns on warnings.

Chapter 6. Perl variables

In this chapter...

In this chapter we will explore Perl's three main variable types --- scalars, arrays, and hashes --- and learn to assign values to them, retrieve the values stored in them, and manipulate them in certain ways. More advanced information about Perl's variables and assignment to them can be found in Appendix A.

What is a variable?

A variable is a place where we can store data. Think of it like a pigeonhole with a name on it indicating what data is stored in it.

The Perl language is very much like human languages in many ways, so you can think of variables as being the "nouns" of Perl. For instance, you might have a variable called "total" or "employee".

Variable names

Variable names in Perl may contain alphanumeric characters in upper or lower case, and underscores. A variable name may not start with a number, as that means something special, which we'll encounter later. Likewise, variables that start with anything non-alphanumeric are also special, and we'll discuss that later, too.

It's standard Perl style to name variables in lower case, with underscores separating words in the name. For instance, `employee_number`. Upper case is usually used for constants, for instance `LIGHT_SPEED` or `PI`. Following these conventions will help make your Perl more maintainable and more easily understood by others.

Lastly, variable names all start with a punctuation sign (correctly known as a *sigil*) depending on what sort of variable they are:

Table 6-1. Variable punctuation

Variable type	Starts with	Pronounced
Scalar	$	dollar
Array	@	at
Hash	%	percent

(Don't worry if those variable type names don't mean anything to you. We're about to cover them.)

Variable scoping and the strict pragma

Many programming languages require you to "pre-declare" variables --- that is, say that you're going to use them before you do so. Variables can either be declared as global (that is, they can be used anywhere in the program) or local (they can only be used in the same part of the program in which

they were declared).

In Perl, it is not necessary to declare your variables before you begin. You can summon a variable into existence simply by using it, and it will be globally available to any routine in your program. If you're used to programming in C or any of a number of other languages, this may seem odd and even dangerous to you. This is indeed the case. That's why you want to use the `strict` pragma.

Arguments in favour of strictness

- avoids accidental creation of unwanted variables when you make a typing error
- avoids scoping problems, for instance when a subroutine uses a variable with the same name as a global variable
- allows for warnings if values are assigned to variables and never used (which is great for detecting typographical errors)

Arguments against strictness

- takes a while to get used to, and may slow down development until it becomes habitual
- enforces a structured style of coding which isn't nearly as much fun

Of course, sometimes a little bit of structure is a good thing, like when you want the trains to run on time. Because of this, Perl lets you turn strictness on if you want it, using something called the *strict pragma*. A pragma, in Perl-speak, is a set of rules for how your code is to be dealt with.


Some documentation about the strict pragma can be found by reading **perldoc strict**. Its effects are discussed on pages 858-860 (page 500 2nd Ed) of the Camel book.

Using the strict pragma (predeclaring variables)

Using strict and warnings will catch the vast majority of common programming errors, and also enforces a more clean and understandable programming style. Following these conventions is also very important if you wish to seek help from other more experienced programmers.

Here's how the strict pragma is invoked:

```
#!/usr/bin/perl -w
use strict;
```

That typically goes at the top of your program, just under your shebang line and introductory comments.

Once we use the strict pragma, we have to explicitly declare new variables using `my`. For example:

```
my $scalar;
my @array;
my %hash;

my $number = 3;
```

These variable declarations can occur anywhere in the program and it is good practice to declare your variables just before you use them. We'll come back to this in more detail when we talk about blocks and subroutines.

 There's more about use of `my` on pages 130-133 (page 189, 2nd Ed) of the Camel book.

Exercise

Try running the program `exercises/strictfail.pl` and see what happens. What needs to be done to fix it? Try it and see if it works. By the way, get used to this error message - it's one of the most common Perl programming mistakes, though it's easily fixed.

An answer for the above can be found at `exercises/answers/strictfail.pl`.

Using the diagnostics pragma

Another pragma that you may find useful is the diagnostics pragma. This translates the normally terse diagnostics emitted from the perl compiler and the perl interpreter into much more useful ones.

To use this pragma, all you have to do is type:

```
use diagnostics;
```

at the start of your code.

The diagnostics pragma makes your warnings much more verbose, and it slows the start-up time of your script considerably. You should remove it before putting your code into production.

 All the extended diagnostics can also be found in **perldoc perldiag**, or in pages 916-978 of the camel book (pages 557-597 2nd Ed).

Further information about the diagnostics pragma can be found by reading **perldoc diagnostics**

Exercise

You can see the diagnostics pragma in action by running the program `exercises/diagnostics.pl`. If you want to, you can remove the `use diagnostics;` line to see the errors without the explanations.

Starting your Perl program

To summarise, your perl program should always start with:

1. A shebang line (with warnings)
2. A comment (what your program does)
3. The strict pragma

For example:

```
#!/usr/bin/perl -w
# This program ......
use strict;
```

You may wish to add `use diagnostics;` while your program is in development.

Scalars

The simplest form of variable in Perl is the scalar. A scalar is a single item of data such as:

- Arthur
- Just Another Perl Hacker
- 42
- 0.000001
- 3.27e17

Here's how we assign values to scalar variables:

```
my $name = "Arthur";
my $whoami = 'Just Another Perl Hacker';
my $meaning_of_life = 42;
my $number_less_than_1 = 0.000001;
my $very_large_number = 3.27e17;          # 3.27 by 10 to the power of 17
```


There are other ways to assign things apart from the = operator, too. They're covered on pages 107-108 (pages 92-93, 2nd Ed) of the Camel book.

A scalar can be text of any length, and numbers of any precision (machine dependent, of course). Perl doesn't need us to tell it what *type* of data we're going to put into the scalar. In fact, Perl doesn't care if the type of data in the scalar changes throughout your program. Perl magically converts between them when it needs to. For instance, it's quite legal to say:

```
# Adding an integer to a floating point number.
my $sum = $meaning_of_life + $number_less_than_1;

# Here we're putting the number in the middle of a string we
# want to print.
print "$name says, 'The meaning of life is $meaning_of_life.'\n";
```

This may seem extraordinarily alien to those used to strictly typed languages, but believe it or not, the ability to transparently convert between variable types is one of the great strengths of Perl. Some people say that it's also one of the great weaknesses.


You can explicitly cast scalars to various specific data types. Look up `int()` on page 731 (page 180, 2nd Ed) of the Camel book, or read **perldoc -f int** for instance.

If you really want strictly typed scalars, Perl lets you have them. Check out **perldoc Attribute::Types**. This isn't installed with Perl by default but can be found at its page on CPAN (http://search.cpan.org/~dconway/Attribute-Types-0.10/lib/Attribute/Types.pm). `Attribute::Types` goes beyond specifying that a given scalar can only hold an integer, for example, as it also allows you to say that it must be between two given values. Alternately you may wish to insist that a string be a member of a selected set or that a value corresponds to the date of a full moon. `Attribute::Types` makes all of these possible. Most Perl programmers don't find this necessary, but sometimes it's invaluable.

Double and single quotes

While we're here, let's look at the assignments above. You'll see that some have double quotes, some have single quotes, and some have no quotes at all.

In Perl, quotes are required to distinguish strings from the language's reserved words or other expressions. Either type of quote can be used, but there is one important difference: double quotes can include other variable names inside them, and those variables will then be interpolated --- as in the print example above --- while single quotes do not interpolate.

```
# single quotes don't interpolate...
my $price = '$9.95';

# double quotes interpolate...
my $invoice_item = "24 widgets at $price each\n";

print $invoice_item;
```

Exercise

The above example is available in your directory as `exercises/interpolate.pl`. Run the script to see what happens. Try changing the type of quotes for each string. What happens?

Special characters

Special characters, such as the `\n` newline character, are only available within double quotes. Single quotes will fail to expand these special characters just as they fail to expand variable names. The only exceptions are that you can quote a single quote or backslash with a backslash.

```
print 'Here\'s an example';
```

When using either type of quotes, you must have a matching pair of opening and closing quotes. If you want to include a quote mark in the actual quoted text, you can escape it by preceding it with a backslash:

```
print "He said, \"Hello!\"\n";
print 'It was Jamie\'s birthday.';
```

You can also use a backslash to escape other special characters such as dollar signs within double quotes:

```
print "The price is \$300\n";
```

To include a literal backslash in a double-quoted string, use two backslashes: `\\`


Perl has other quoting structures to help you avoid having to escape your quotes continually. To read more about these, look at Appendix A.

Advanced Variable Interpolation

Sometimes you'll want to do something like the following:

```
my $what = "jumper";
print "I have 4 $whats";
```

but this won't work, because there is no such variable `$whats`, or if there is, it's probably not the one we want to be using. We could do:

```
my $what = "jumper";
print "I have 4 " . $what . "s";
```

and if you like that, then it's fine. However, that's pretty ugly, and there's a nicer looking way of doing it which involves less keystrokes as well:

```
my $what = "jumper";
print "I have 4 ${what}s";
```

I'm sure that you'll agree that's much better.


There are special quotes for executing a string as a shell command (see "Input operators" on page 79 (page 52, 2nd Ed) of the Camel book), and also special quoting functions (see "Pick your own quotes" on page 63 (page 41, 2nd Ed)). These are also covered in Appendix A.

Exercises

1. Write a script which sets some variables:
    a. your name
    b. your street number
    c. your favourite colour
2. Print out the values of these variables using double quotes for variable interpolation.
3. Change the quotes to single quotes. What happens?
4. Write a script which prints out the string `C:\WINDOWS\SYSTEM\` twice -- once using double quotes, once using single quotes. How do you have to escape the backslashes in each case?

You'll find answers to the above in `exercises/answers/scalars.pl`.

Arrays

If you think of a scalar as being a singular thing, arrays are the plural form. Just as you have a flock of chickens or a wunch of bankers, you can have an array of scalars.

An array is a list of (usually related) scalars all kept together. Arrays start with an `@` (at sign).

 Arrays are discussed on pages 9-10 (page 6, 2nd Ed) of the Camel book and also in **perldoc perldata**.

Initialising an array

Arrays are initialised by creating a comma separated list of values:

```
my @fruits = ("apples", "oranges", "guavas", "passionfruit", "grapes");
my @magic_numbers = (23, 42, 69);
my @random_scalars = ("mumble", 123.45, "willy the wombat", -300);
```

As you can see, arrays can contain any kind of scalars. They can have just about any number of elements, too, without needing to know how many before you start. *Really* any number - tens or hundreds of thousands, if your computer has the memory.

Reading and changing array values

First of all, Perl's arrays, like those in many other languages, are indexed from zero. We can access individual elements of the array like this:

```
print $fruits[0];               # prints "apples"
print $random_scalars[2];       # prints "willy the wombat"
$fruits[0] = "pineapples";      # Changes "apples" to "pineapples"
```

Wait a minute, why are we using dollar signs in the example above, instead of at signs? The reason is this: we only want a scalar back, so we show that we want a scalar. There's a useful way of thinking of this, which is explained in chapter 1 (both editions) of the Camel book: if scalars are the singular case, then the dollar sign is like the word "the" - "the name", "the meaning of life", etc. The `@` sign on an array, or the `%` sign on a hash, is like saying "those" or "these" - "these fruit", "those magic numbers". However, when we only want one element of the array, we'll be saying things like "the first fruit" or "the last magic number" - hence the scalar-like dollar sign.

Array slices

If we wanted to only deal with a portion of the array, we use what we call an *array slice*. These are written as follows:

```
@fruits[1,2,3];         # oranges, guavas, passionfruit
@fruits[3,1,2];         # passionfruit, oranges, guavas
@magic_numbers[0..2];   # 23, 42, 69
@magic_numbers[1..5] = (46, 19, 88, 12, 23);  # Assigns new magic numbers
```

You'll notice that these array slices have `@` signs in front of them. That's because we're still dealing with a list of things, just one that's (typically) smaller than the full array. It is possible to take an array slice of a single element:

```
@fruits[1];             # array slice of one element
```

but this usually means that you've made a mistake and Perl will warn you that what you really should be writing is:

```
$fruits[1];
```

You just learnt something new back there: the `..` ("dot dot") range operator creates a temporary list of numbers between the two you specify. In our case we specified 0 and 2 (then 1 and 5), but it could have been 1 to 100, or 30 to 70, if we'd had an array big enough to use it on. You'll run into this operator again and again.


See pages 103-104 (pages 90-91, 2nd Ed) of the Camel book or **perldoc perlop**) for more information about the dot dot operator.

Array interpolation

Another thing you can do with arrays is insert them into a string, the same as for scalars:

```
print "My favourite fruits are @fruits\n";      # whole array
print "Two types of fruit are @fruits[0,2]\n";  # array slice
print "My favourite fruit is $fruits[3]\n";     # single element
```

Counting backwards

It's also possible to count backwards from the end of an array, like this:

```
$fruits[-1];     # Last fruit in the array, grapes in this case.
$fruits[-3];     # Third last fruit: guavas.
```

Finding out the size of an array

So if we don't know how many items there are in an array, how can we find out? Well, there's a special syntax `$#array` which is the index of the last element, so you can say:

```
my $last = $#fruits;       # index of last element
@fruits[($last/2) .. $last];
```

and you'll get the second half of the array. If you print `$#fruits` you'll find it's 4, which is not the same as the number of elements: 5. Remember that it's the *index of the last element* and that the index *starts at zero*, so you have to add one to it to find out how many elements are in the array.

```
my $number_of_fruits = $#fruits + 1;
```

If the array is empty, `$#fruits` returns `-1`.

But wait! There's More Than One Way To Do It - and an easier way, at that. If you evaluate the array in a scalar context - that is, do something like this:

```
my $fruit_count = @fruits;
```

you'll get the number of elements in the array.

There's more than two ways to do it, as well --- `scalar(@fruits)` and `int(@fruits)` will also tell us how many elements there are in the array. Both of these functions force a scalar context, so they're really using the same mechanism as the `$fruit_count` example above. We'll talk more about contexts soon.

Using qw// to populate arrays

If you're working with lists and arrays a lot, you might find that it gets very tiresome to be typing so many quotes and commas. Let's take our fruit example:

```
my @fruits = ("apples", "oranges", "guavas", "passionfruit", "grapes");
```

We had to type the quotes character ten times, along with four commas, and that was only for a short list. If your list is longer, such as all the months in a year, then you'll need even more punctuation to make it all work. It's easy to forget a quote, or use the wrong quote, or misplace a comma, and end up with a trivial but bothersome error. Wouldn't it be nice if there was a better way to create a list?

Well, there is a better way, using the `qw//` operator. `qw//`stands for *quote words*. It takes whitespace separated words and turns them into a list, saving you from having to worry about all that tiresome punctuation. Here's our fruit example again using `qw//`:

```
my @fruits = qw/apples oranges guavas passionfruit grapes/;
```

As you can see, this is clear, concise, and difficult to get wrong. And it keeps getting better. Your list can stretch over multiple lines, and your delimiter doesn't need to be a slash. Whatever punctuation character that you place after the `qw` becomes the delimiter. So if you prefer parentheses over slashes, that's no problem at all:

```
my @months = qw(January February March April May June July August
                September October November December);
```

For more information about `qw//` and other quoting mechanisms, see see "Pick your own quotes" on page 63 (page 41, 2nd Ed) of the Camel book. There's also an excellent discussion in **perldoc perlop** in the *Quote and Quote-like Operators* section. This is also covered in Appendix A.

Printing out the values in an array

If you want to print out all the values in an array there are several ways you can do it:

```
my @fruits = qw/apples oranges guavas passionfruit grapes/;
print @fruits;            # prints "applesorangesguavaspassionfruitgrapes"
print join(", ", @fruits);# prints "apples, oranges, guavas, passionfruit, grapes"
print "@fruits";          # prints "apples oranges guavas passionfruit grapes"
```

The first method takes advantage of the fact that print takes a list of arguments to print and prints them out sequentially. The second uses `join()` which joins an array or list together by separating

each element with the first argument. The third option uses double quote interpolation and a little bit of Perl magic to pick which character(s) to separate the words with.

A quick look at context

There's a term you've heard used just recently but which hasn't been explained: *context*. Context refers to how an expression or variable is evaluated in Perl. The two main contexts are:

- scalar context, and
- list context

Scalar variables are always evaluated in scalar context, however arrays and hashes can be evaluated in both scalar contexts (when we treat them as scalars) and list contexts (when we treat them as arrays and hashes).

Here's an example of an expression which can be evaluated in either context:

```
my @newarray = @array;          # list context
my $howmany  = @array;          # scalar context
my $howmany2 = scalar(@array);  # scalar context again (explicitly)
```

If you look at an array in a scalar context, you'll see how many elements it has; if you look at it in list context, you'll see the contents of the array itself.

Many things in Perl are very specific about which context they require and will *force* lists into scalar context when required. For example the `+` (plus or addition) operator expects that its two arguments will be scalars. Hence:

```
my @a = (qw/1 2 3/);
my @b = (qw/4 5 6 7/);

print (@a + @b);
```

will print `7` (the sum of the two list's lengths) rather than `5 7 9 7` (the individual sums of the list elements).

Many things in Perl have different behaviours depending upon whether or not they're in an array or scalar context. This is generally considered a good thing, as it means things can have a "Do What I Mean" (DWIM) behaviour depending upon how they are used. Arrays are the most common example of this, but we'll see some more as we progress through the course.

There's also a third type of context, the null context, where the result of an operation is just thrown away. This usually isn't discussed, because by its very definition we don't care about what result is returned.

What's the difference between a list and an array?

Not much, really. A list is just an unnamed array. Here's a demonstration of the difference:

```
# printing a list of scalars
print ("Hello", " ", $name, "\n");
```

```
# printing an array
my @hello = ("Hello", " ", $name, "\n");
print @hello;
```

If you come across something that wants a LIST, you can either give it the elements of list as in the first example above, or you can pass it an array by name. If you come across something that wants an ARRAY, you have to actually give it the name of an array. Examples of functions which insist on wanting an ARRAY are `push()` and `pop()`, which can be used for adding and removing elements from the end of an array.

 List values and Arrays are covered on page 72 (page 47, 2nd Ed) of the Camel book.

Exercises

1. Create an array of your friends' names. (You're encouraged to use the `qw//` operator.)
2. Print out the first element.
3. Print out the last element.
4. Print the array within a double-quoted string, ie: `print "@friends";` and notice how Perl handles this.
5. Print out an array slice of the 2nd to 4th items within a double-quoted string (variable interpolation).
6. Replace every second friend with another friend.
7. Write print statement to print out your email address. How can you handle the `@` when you're using double quotes?

Answers to the above can be found in `exercises/answers/arrays.pl`

Advanced exercises

1. Print the array without putting quotes around its name, ie: `print @friends;`. What happens? How is this different from what happens, when you printed the array enclosed in double quotes?
2. What happens if you have a small array and then you assign a value to `$array[1000]`? Print out the array.

Answers to the above can be found in `exercises/answers/arrays_advanced.pl`

Hashes

A hash is a two-dimensional array which contains keys and values, they're sometimes called "associative arrays", or "lookup tables". Instead of looking up items in a hash by an array index, you can look up values by their keys.

To find out more about hashes and hash slices have a look at Appendix A.

 Hashes are covered in the Camel book on pages 6-10 (pages 7-8, 2nd Ed), then in more detail on pages 76-78 (page 50, 2nd Ed) or in **perldoc perldata**.

Initialising a hash

Hashes are initialised in exactly the same way as arrays, with a comma separated list of values:

```
my %monthdays = ("January", 31, "February", 28, "March", 31, ...);
```

Of course, there's more than one way to do it:

```
my %monthdays = (
        "January"       =>      31,
        "February"      =>      28,
        "March"         =>      31,
        ...
);
```

The spacing in the above example is commonly used to make hash assignments more readable.

The => operator is syntactically the same as the comma, but is used to distinguish hashes more easily from normal arrays. It's pronounced "fat comma", or less often "fat arrow". It does have one difference, you don't need to put quotes around a bare word immediately before the => operator as these are always treated as strings:

```
my %monthdays = (
        January       =>      31,
        February      =>      28,
        March         =>      31,
        ...
);

        # Note that we still have to quote strings on the right hand side.
my %pizza_prices = (
        small         => '$5.00',
        medium        => '$7.50',
        large         => '$9.00',
);
```

Reading hash values

You get at elements in a hash by using the following syntax:

```
print $monthdays{"January"};    # prints 31
```

Again you'll notice the use of the dollar sign, which you should read as "the monthdays value belonging to January".

Bare words inside the braces of the hash-lookup will be interpreted in Perl as strings, so usually you can drop the quotes:

```
print $monthdays{March};        # prints 31
```

Adding new hash elements

You can also create elements in a hash on the fly:

```
my %monthdays = ();
$monthdays{"January"} = 31;
$monthdays{February} = 28;
...
```

Changing hash values

To change a value in a hash, just assign the new value to your key:

```
$pizza_prices{small} = '$6.00';      # Small pizza prices have gone up
```

Deleting hash values

To delete an element from a hash you need to use the `delete` function. This is used as follows:

```
delete($pizza_prices{medium});          # No medium pizzas anymore.
```

Finding out the size of a hash

Strictly speaking there is no equivalent to using an array in a scalar context to get the size of a hash. If you take a hash in a scalar context you get back the number of buckets used in the hash, or zero if it is empty. This is only really useful to determine whether or not there are any items in the hash, not how many.

If you want to know the size of a hash, the number of key-value pairs, you can use the `keys` function in a scalar context. The `keys` function returns a list of all the keys in the hash.

```
my $size = keys %monthdays;
print $size;                    # prints "12" (so long as the hash contains
                                # all 12 months)
```

Other things about hashes

- Hashes have no internal order.
- There are functions such as `each()`, `keys()` and `values()` which will help you manipulate hash data. We look at these later, when we deal with functions.
- Hash lookup is very fast, and is the speediest way of storing data that you need to access in a random fashion.

You may like to look up the following functions which related to hashes: `keys()`, `values()`, `each()`, `delete()`, `exists()`, and `defined()`. You can do that using the command **perldoc -f *`function-name`***.

Exercises

1. Create a hash of people and something interesting about them.
2. Print out a given person's interesting fact.
3. Change a person's interesting fact.
4. Add a new person to the hash.
5. What happens if you try to print an entry for a person who's not in the hash?
6. What happens if you try to print out the hash outside of any quotes? Look at the order of the elements.
7. What happens if you try to print out the hash inside double quotes? Do you understand why this happens?
8. What happens if you attempt to assign an array as a value into your hash?

Answers to these exercises are given in `exercises/answers/hash.pl`

Special variables

Perl has many special variables. These are used to set or retrieve certain values which affect the way your program runs. For instance, you can set a special variable to turn interpreter warnings on and off (`$^W`), or read a special variable to find out the command line arguments passed to your script (`@ARGV`).

Special variables can be scalars, arrays, or hashes. We'll look at some of each kind.


Special variables are discussed at length in chapter 2 of your Camel book (from page 653 (page 127, 2nd Ed) onwards) and in the `perlvar` manual page. You may also like to look up the `English` module, which lets you use longer, more English-like names for special variables. You'll find more information on this by using **perldoc English** to read the module documentation.

Special variables don't need to be declared like regular variables, as Perl already knows they exist. In fact, it's an error to try and declare a special variable with `my`.


Changing a special variable in your code changes it for the entire program, from that point onwards.

The special variable $_

The special variable that you'll encounter most often, is called `$_` ("dollar-underscore"), and it represents the current thing that your Perl script's working with --- often a line of text or an element of a list or hash. It can be set explicitly, or it can be set implicitly by certain looping constructs (which we'll look at later).

The special variable `$_` is often the default argument for functions in Perl. For instance, the `print()` function defaults to printing `$_`.

```
$_ = "Hello, world!\n";
print;
```

If you think of Perl variables as being "nouns", then `$_` is the pronoun "it".

 There's more discussion of using `$_` on page 658 (page 131, 2nd Ed) of your Camel book.

Exercises

1. Set `$_` to a string like "Hello, world", then print it out by using the `print()` command's default argument

@ARGV - a special array

Perl programs accept arbitrary arguments or parameters from the command line, like this:

```
% printargs.pl foo bar baz
```

This passes "foo", "bar" and "baz" as arguments into our program, where they end up in an array called `@ARGV`.

Exercise

- The **printargs** script mentioned in the above example can be found in `exercises/printargs.pl`. Run this script now passing in some arguments.
- Write a program which takes two arguments from the command line (a name and a favourite food) and then prints out a message detailing that name likes that food. An answer can be found in `exercises/answers/favouritefood.pl`

%ENV - a special hash

Just as there are special scalars and arrays, there is a special hash called `%ENV`. This hash contains the names and values of environment variables. For example, the value of the environment variable USER is available in `$ENV{"USER"}`. To view these variables under Unix, simply type **env** on the command line. To view these under Microsoft Windows type **set**.

Exercises

1. A user's home directory is stored in the environment variable `HOME` (Unix) or `HOMEPATH` (MS Windows). Print out your own home directory.
2. What other things can you find in `%ENV`?

The answer to the above can be found in `exercises/answers/env.pl`

Changing a value in the %ENV hash changes your program's current environment. Any changes to your program environment will be inherited by any child processes your program invokes. However you cannot change the environment of the shell from which your program is called.

Chapter summary

- Perl variable names typically consist of alphanumeric characters and underscores. Lower case names are used for most variables, and upper case for global constants.
- The statement `use strict;` is used to make Perl require variables to be pre-declared and to avoid certain types of programming errors.
- There are three types of Perl variables: scalars, arrays, and hashes.
- Scalars are single items of data and are indicated by a dollar sign (`$`) at the beginning of the variable name.
- Scalars can contain strings, numbers and references.
- Strings must be delimited by quote marks. Using double quote marks will allow you to interpolate other variables and meta-characters such as `\n` (newline) into a string. Single quotes do not interpolate.
- Arrays are one-dimensional lists of scalars and are indicated by an at sign (`@`) at the beginning of the variable name.
- Arrays are initialised using a comma-separated list of scalars inside round brackets.
- Arrays are indexed from zero
- Item `n` of an array can be accessed by using `$arrayname[n]`.
- The index of the last item of an array can be accessed by using `$#arrayname`.
- The number of elements in an array can be found by interpreting the array in a scalar context, eg `my $items = @array;`
- Hashes are two-dimensional arrays of keys and values, and are indicated by a percent sign (`%`) at the beginning of the variable name.
- Hashes are initialised using a comma-separated list of scalars inside curly brackets. Whitespace and the => operator (which is syntactically identical to the comma) can be used to make hash assignments look neater.
- The value of a hash item whose key is `foo` can be accessed by using `$hashname{foo}`
- Hashes have no internal order.
- `$_` is a special variable which is the default argument for many Perl functions and operators
- The special array `@ARGV` contains all command line parameters passed to the script
- The special hash `%ENV` contains information about the user's environment.

Chapter 7. Operators and functions

In this chapter...

In this chapter, we look at some of the operators and functions which can be used to manipulate data in Perl. In particular, we look at operators for arithmetic and string manipulation, and many kinds of functions including functions for scalar and list manipulation, more complex mathematical operations, type conversions, dealing with files, etc.

What are operators and functions?

Operators and functions are routines that are built into the Perl language to do stuff.

The difference between operators and functions in Perl is a very tricky subject. There are a couple of ways to tell the difference:

- Functions usually have all their parameters on the right hand side,
- Operators can act in much more subtle and complex ways than functions,
- Look in the documentation --- if it's in **perldoc perlop**, it's an operator; if it's in **perldoc perlfunc**, it's a function. Otherwise, it's probably a subroutine.

The easiest way to explain operators is to just dive on in, so here we go.

Operators

There are lists of all the available operators, and what they each do, on pages 86-110 (pages 76-94, 2nd Ed) of the Camel book. You can also see them by typing **perldoc perlop**. Precedence and associativity are also covered there.

If you've programmed in C before, then most of the Perl operators will be already be familiar to you. Perl operators have the same precedence as they do in C. Perl also adds a number of new operators which C does not have.

Arithmetic operators

Arithmetic operators can be used to perform arithmetic operations on variables or constants. The commonly used ones are:

Table 7-1. Arithmetic operators

Operator	Example	Description
+	`$a + $b`	Addition
-	`$a - $b`	Subtraction

Operator	Example	Description
*	$a * $b	Multiplication
/	$a / $b	Division
%	$a % $b	Modulus (remainder when $a is divided by $b, eg 11 % 3 = 2)
**	$a ** $b	Exponentiation ($a to the power of $b)


Just like in C, there are some short cut arithmetic operators:

```
$a += 1;        # same as $a = $a + 1
$a -= 3;        # same as $a = $a - 3
$a *= 42;       # same as $a = $a * 42
$a /= 2;        # same as $a = $a / 2
$a %= 5;        # same as $a = $a % 5;
$a **= 2;       # same as $a = $a ** 2;
```

(In fact, you can extrapolate the above with just about any operator --- see page 26 (page 17, 2nd Ed) of the Camel book for more about this).

You can also use `$a++` and `$a--` if you're familiar with such things. `++$a` and `--$a` also exist, which increment (or decrement) the variable before evaluating it.

For example:

```
my $a = 0;
print $a++;     # prints "0", but sets $a to 1.
print ++$a;     # prints "2" after it has set $a to 2.
```

String operators

Just as we can add and multiply numbers, we can also do similar things with strings:

Table 7-2. String operators

Operator	Example	Description
.	$a . $b	Concatenation (puts $a and $b together as one string)
x	$a x $b	Repeat (repeat $a $b times --- eg "foo" x 3 gives us "foofoofoo"


These can also be used as short cut operators:

```
$a .= " foo";   # same as $a = $a . " foo";
$a .= $bar;     # same as $a = $a . $bar;
$a x= 3;        # same as $a = $a x 3;
```


There's more about the concatenation operator on page 95 (page 16, 2nd Ed) of the Camel book.

Exercises

1. Calculate the cost of 17 widgets at $37.00 each and print the answer. (Answer: `exercises/answers/widgets.pl`)
2. Print out a line of dashes without using more than one dash in your code (except for the `-w`). (Answer: `exercises/answers/dashes.pl`)
3. Look over `exercises/operate.pl` for examples on how to use arithmetic and string operators.

File test operators

We can use file test operators to test various attributes of files and directories. If you've programmed in the Bourne Shell before, then many of these file test operators will be familiar to you.

Table 7-3. File test operators

Operator	Example	Description
`-e`	`-e $a`	Exists - does the file exist?
`-r`	`-r $a`	Readable - is the file readable?
`-w`	`-w $a`	Writable - is the file writable?
`-d`	`-d $a`	Directory - is it a directory?
`-f`	`-f $a`	File - is it a normal file?
`-T`	`-T $a`	Text - is the file a text file?


There are examples of these in use on pages 99-100 (pages 19-20, 2nd Ed) of the Camel book. There is a complete list of the file operators in the Camel book on page 98 (page 85, 2nd Ed), or in **perldoc -f -x**. There are lots!

These file test operators are extremely easy to use. For example if we want to do something only if a file exists (let's say "/etc/passwd") all we need type is:

```
if( -e "/etc/passwd") {
        # do stuff.
}
```

We can also do several tests:

```
if( -r "testfile" and -w "testfile") {
        # The file is available for reading and writing.
}
```

Now this assumes that you're comfortable with conditional (if) statements, never mind if you're not, we'll be covering them soon.

Other operators

You'll encounter all kinds of other operators in your Perl career, and they're all described in the Camel book from page 86 (page 76, 2nd Ed) onwards. We'll cover them as they become necessary to us -- you've already seen operators such as the assignment operator (=), the fat comma operator (=>) which behaves a bit like a comma, and so on.

While we're here, let's just mention what "unary" and "binary" operators are.

A unary operator is one that only needs something on one side of it, like the file operators or the auto-increment (++) operator.

A binary operator is one that needs something on either side of it, such as the addition operator.

A trinary operator also exists, but we don't deal with it in this course. C programmers will probably already know about it, and can use it if they want.

Functions

There's an introduction to functions on page 16 (page 8, 2nd Ed) of the Camel book, labelled 'Verbs'. Check out **perldoc perlfunc** too.

To find the documentation for a single function you can use **perldoc -f *`functionname`***. For example **perldoc -f print** will give you all the documentation for the `print` function.

A function is like an operator --- and in fact some functions double as operators in certain conditions --- but with the following differences:

- longer names which are words rather than punctuation,
- can take any types of arguments,
- arguments always come *after* the function name.

The only real way to tell whether something is a function or an operator is to check the `perlop` and `perlfunc` manual pages and see which it appears in.

We've already seen and used a very useful function: `print`. The print function takes a list of arguments (to print). For example:

```
my @array = qw/Buffy Willow Xander Giles/;
print @array;   # print taking an array (which is just a named list).
print "@array"; # Variable interpolation in our string adds spaces
                # between elements.

print "Willow and Xander";      # Printing a single string.
print("Willow and Xander");     # The same.
```

As you'll have noticed, Perl does not insist that functions enclose their arguments within parentheses. Both `print "Hello";` and `print("Hello");` are correct. Feel free to use parentheses if you want to. It usually makes your code easier to read.

There are good reasons for using parentheses all the time, because it's easy to make certain mistakes if you don't. Take the following example:

```
print (3 + 7) * 4;      # Wrong!
```

This prints 10, not 40. The reason is that whenever any Perl function sees parentheses after a function or subroutine name, it presumes that to be its argument list. So Perl has interpreted the line above as:

```
(print(3+7) )*4;
```

That's almost certainly not what you wanted. In fact, if you forgot to turn warnings on, it would almost certainly provide you with many hours of fruitless debugging.

The best way of getting around this is to always use parentheses around your arguments:

```
print ((3+7) * 4);       # Correct!
```

As you become more experienced with Perl, you can learn when it's safe to drop the parentheses.

Types of arguments

Functions typically take the following kind of arguments:

SCALAR -- Any scalar variable, for example: `42`, `"foo"`, or `$a`.

EXPR -- An expression (possibly built out of terms and operators) which evaluates to a scalar.

LIST -- Any named or unnamed list (remember that a named list is an array).

ARRAY -- A named list; usually results in the array being modified.

HASH -- Any named or unnamed hash.

PATTERN -- A pattern to match on --- we'll talk more about these later on, in Regular Expressions.

FILEHANDLE -- A filehandle indicating a file that you've opened or one of the pseudo-files that is automatically opened, such as STDIN, STDOUT, and STDERR.

There are other types of arguments, but you're not likely to need to deal with them in this module.

In chapter 29 (chapter 3, 2nd Ed) of the Camel book (starting on page 677, or page 141 2nd Ed), you'll see how the documentation describes what kind of arguments a function takes.

Return values

Just as functions can take arguments of various kinds, they can also return values which you can use. The simplest return value is nothing at all, although this is rare for Perl functions. Functions typically return scalars or lists which you can; use immediately, capture for later or ignore.

If a function returns a scalar, and we want to use it, we can say something like:

```
my $age = 29.75;
my $years = int($age);
```

and `$years` will be assigned the returned value of the `int()` function when given the argument `$age` --- in this case, 29, since `int()` truncates instead of rounding.

If we just wanted to do something to a variable and didn't care what value was returned, we can call the function without looking at what it returns. Here's an example:

```
my $input = <STDIN>;
chomp($input);
```

`chomp`, as you'll see if you type **perldoc -f chomp**, is typically used to remove the newline character from the end of the arguments given to it. `chomp` returns the number of characters removed from all of its arguments. `<STDIN>` takes a line from STDIN (usually the keyboard). We talk more about this later.

Functions can also return arrays and hashes, as well as scalars. For example, the `sort` function returns a sorted array:

```
@sorted = sort @array;
```

More about context

We mentioned earlier a few things about *list context* and *scalar context*, and how arrays act differently depending upon how you treat them. Functions and operators are the same. If a function or operator acts differently depending upon context, it will be noted in the Camel book and the manual pages.

Here are some Perl functions that care about context:

Table 7-4. Context-sensitive functions

What?	Scalar context	List context
`reverse()`	Reverses characters in a string.	Reverses the order of the elements in an array.
`each()`	Returns the next key in a hash.	Returns a two-element list consisting of the next key and value pair in a hash.
`gmtime()` and `localtime()`	Returns the time as a string in common format.	Returns a list of second, minute, hour, day, etc.
`keys()`	Returns the number of keys (and hence the number of key-value pairs) in a hash.	Returns a list of all the keys in a hash.
`readdir()`	Returns the next filename in a directory, or undef if there are no more.	Returns a list of all the filenames in a directory.

There are many other cases where an operation varies depending on context. Take a look at the notes on context at the start of **perldoc perlfunc** to see the official guide to this: "anything you want, except consistency".

Some easy functions

 Starting on page 683 (page 143, 2nd Ed) of the Camel book, there is a list of every single Perl function, their arguments, and what they do. These are also listed in **perldoc perlfunc**.

String manipulation

Finding the length of a string

The length of a string can be found using the `length()` function:

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

use strict;

my $string = "This is my string";
print length($string);
```

Case conversion

You can convert Perl strings from upper case to lower case, or vice versa, using the `lc()` and `uc()` functions, respectively.

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

print lc("Hello, World!");              # prints "hello, world!"
print uc("Hello, World!");              # prints "HELLO, WORLD!"
```

The `lcfirst()` and `ucfirst()` functions can be used to change only the first letter of a string.

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

print lcfirst("Hello, World!");         # prints "hello, World!"
print ucfirst(lc("Hello, World!"));     # prints "Hello, world!"
```

Notice how, in the last line of the example above, the `ucfirst()` operates on the result of the `lc()` function.

chop() and chomp()

The `chop()` function removes the last character of a string and returns that character.

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

use strict;
my $string = "Goodbye";

my $char = chop $string;
print $char;                            # "e"
print $string;                          # "Goodby"
```

The `chomp()` function works similarly, but *only* removes the last character if it is a newline. It will only remove a single newline per string. `chomp()` returns the number of newlines it removed, which

will be 0 or 1 in the case of chomping a single string. chomp() is invaluable for removing extraneous newlines from user input.

```
#!/usr/bin/perl -w
use strict;

my $string1 = "let's go dancing!";
my $string2 = "dancing, dancing!\n";

my $chomp1 = chomp $string1;
my $chomp2 = chomp $string2;

print $string1;                         # "let's go dancing!";
print $string2;                         # "dancing, dancing!";

print $chomp1;                          # 0  (there was no newline to remove)
print $chomp2;                          # 1  (removed one newline)
```

Both chop and chomp can take a list of things to work on instead of a single element. If you chop a list of strings, only the value of the last chopped character is returned. If you chomp a list, the total number of characters removed is returned.

Actually, chomp removes any trailing characters that correspond to the input record separator ($/), which is a newline by default. This means that chomp is very handy if you're reading in records which are separated by known strings, and you want to remove your separators from your records.

String substitutions with substr()

The substr() function can be used to return a portion of a string, or to change a portion of a string. substr takes up to four arguments:

1. The string to work on.
2. The offset from the beginning of the string from which to start the substring. (First character has position zero).
3. The length of the substring. Defaults to be from offset to end of the string.
4. String to replace the substring with. If not supplied, no replacement occurs.

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

use strict;

my $string = "*** Hello, world! ***\n";
print substr($string, 4, 5);            # prints "Hello"

substr($string, 4, 5) = "Greetings";
print $string;                          # prints "*** Greetings, world! ***"

substr($string, 4, 9, "Howdy");
print $string;                          # prints "*** Howdy, world! ***"
```

Exercises

1. Create a scalar variable containing the phrase "There's more than one way to do it" then print it out in all upper-case. (Answer: `exercises/answers/tmtowtdi.pl`)
2. Print out the third character of a word entered by the user as an argument on the command line. (There's a starter script in `exercises/thirdchar.pl` and the answer's in `exercises/answers/thirdchar.pl`)
3. Create a scalar variable containing the string "The quick brown fox jumps over the lazy dog". Print out the length of this string, and then using substr, print out the fourth word (fox). (Answer: `exercises/answers/substr.pl`)
4. Replace the word "fox" in the above string with "kitten".

Numeric functions

There are many numeric functions in Perl, including trigonometric functions and functions for dealing with random numbers. These include:

- `abs()` (absolute value)
- `cos()`, `sin()`, and `atan2()`
- `exp()` (exponentiation)
- `log()` (logarithms)
- `rand()` and `srand()` (random numbers)
- `sqrt()` (square root)

Type conversions

The following functions can be used to force type conversions (if you really need them):

- `oct()`
- `int()`
- `hex()`
- `chr()`
- `ord()`
- `scalar()`

Manipulating lists and arrays

Stacks and queues

Stacks and queues are special kinds of lists.

A stack can be thought of like a stack of paper on a desk. Things are put onto the top of it, and taken off the top of it. Stacks are also referred to as "LIFO" (for "Last In, First Out").

A queue, on the other hand, has things added to the end of it and taken out of the start of it. Queues are also referred to as "FIFO" lists (for "First In, First Out").

We can simulate stacks and queues in Perl using the following functions:

- `push()` -- add items to the end of an array.
- `pop()` -- remove items from the end of an array.
- `shift()` -- remove items from the start of an array.
- `unshift()` -- add items to the start of an array.

A queue can be created by `push`ing items onto the end of an array and `shift`ing them off the front.

A stack can be created by `push`ing items on the end of an array and `pop`ping them off.

```
my @stack = qw(a b c d e f g);
my @queue = qw(1 2 3 4 5 6 7 8 9 10);

        # Pop something off the stack:
my $current = pop @stack;       # stack is now: a b c d e f

        # Push something on to the stack:
push @stack, 'h';               # stack is now: a b c d e f h

        # Push something on to the queue:
push @queue, '11';              # queue is now: 1 2 3 4 5 6 7 8 9 10 11

        # Shift something off the queue:
my $next = shift @queue;        # queue is now: 2 3 4 5 6 7 8 9 10 11
```

Ordering lists

The `sort()` function, when used on a list, returns a sorted version of that list. It *does not* alter the original list.

The `reverse()` function, when used on a list, returns the list in reverse order. It *does not* alter the original list.

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

my @list = ("a", "z", "c", "m");
my @sorted = sort(@list);
my @reversed = reverse(sort(@list));
```

Converting lists to strings, and vice versa

The `join()` function can be used to join together the items in a list into one string. Conversely, `split()` can be used to split a string into elements for a list.

```
#!/usr/bin/perl -w
use strict;

my $record = "Fenwick:Paul:Melbourne:Australia";
my @fields = split(/:/,$record);

# @fields is now ("Fenwick","Paul","Melbourne","Australia");

my $newrecord = join(",",@fields);

# $newrecord is now "Fenwick,Paul,Melbourne,Australia";
```

The `/:/` in the split function is a *regular expression*. It tells split what it should split on. We'll cover regular expressions in more details later.

Using `split` and `join` in simple cases such as the above is fine. However often real world data is much more complicated, such as comma or tab separated files, where the separator may be allowed within the fields as well. For such tasks, we recommend the use of the `Text::CSV_XS` module from CPAN.

Exercises

These exercises range from easy to difficult. Answers are provided in the exercises directory (filenames are given with each exercise).

1. Using `split`, print out the fourth word of the string "The quick brown fox jumps over the lazy dog".
2. Print a random number.
3. Print a random item from an array. (Answer: `exercises/answers/quotes.pl`)
4. Print out a sentence in reverse
   a. reverse the whole sentence (eg, `ecnetnes elohw eht esrever`).
   b. reverse just the words (eg, `words the just reverse`).

   (Answer: `exercises/answers/reverse.pl`) Hint: You may find `split` (**perldoc -f split**) useful for this exercise.
5. Write a program which takes words on the command line and prints them out in a sorted order. Change your sort method from asciibetical to alphabetical. Hint: you may wish to read **perldoc -f sort** to see how you can pass your own comparison to the sort function. (Answer: `exercises/answers/command_sort.pl`)
6. Add and remove items from an array using `push`, `pop`, `shift` and `unshift`. (Answer: `exercises/answer/pushpop.pl`)

Hash processing

The `delete()` function deletes an element from a hash.

The `exists()` function tells you whether a certain key exists in a hash.

The `keys()` and `values()` functions return lists of the keys or values of a hash, respectively.

The `each()` function allows you to iterate over key-value pairs.

Reading and writing files

The `open()` function can be used to open a file for reading or writing. The `close()` function closes a file after you're done with it.

We cover reading from and writing to files later in the course. These are not covered further here.

Time

The `time()` function returns the current time in Unix format (that is, the number of seconds since 1 Jan 1970).

The `gmtime()` and `localtime()` functions can be used to get a more friendly representation of the time, either in Greenwich Mean Time or the local time zone. Both can be used in either scalar or list context.

Exercises

1. Create a hash and `delete` an element. Use `exists` to test if hash keys do or do not exist. (Answer: `exercises/answers/hash2.pl`)
2. Print the list of keys in a hash. (Answer: `exercises/answers/hash2.pl`)
3. Print out the date for a week ago (the answer's in `exercises/answers/lastweek.pl`)
4. Read **perldoc -f localtime**.

Chapter summary

- Perl operators and functions can be used to manipulate data and perform other necessary tasks.
- The difference between operators and functions is blurred; most can behave in either way.
- Functions can accept arguments of various kinds.
- Functions may return any data type.
- Return values may differ depending on whether a function is called in scalar or list context.

Chapter 8. Conditional constructs

In this chapter...

In this chapter, we look at Perl's various conditional constructs and how they can be used to provide flow control to our Perl programs. We also learn about Perl's meaning of truth and how to test for truth in various ways.

What is a conditional statement?

A conditional statement is one which allows us to test the truth of some condition. For instance, we might say "If the ticket price is less than ten dollars..." or "While there are still tickets left..."

You've almost certainly seen conditional statements in other programming languages, but Perl has a conditional that you probably haven't seen before. The `unless(`*`condition`*`)` is exactly the same as `if(!(`*`condition`*`))` but easier to read.

 Perl's conditional statements are listed and explained on pages 111-115 (pages 95-106, 2nd Ed) of the Camel book.

What is truth?

Conditional statements invariably test whether something is true or not. Perl thinks something is true if it doesn't evaluate to the number zero (`0`), the string containing a single zero (`"0"`), an empty string (`""`), or the undefined value.

```
""              # false (" would also be false)
42              # true
0               # false
"0"             # false
"00"            # true, only a single zero is considered false
"wibble"        # true
$new_variable   # false (if we haven't set it to anything, it's undefined)
```

 The Camel book discusses Perl's idea of truth on pages 29-30 (pages 20-21, 2nd Ed) including some odd cases.

The if conditional construct

A very common conditional construct is to say: if this thing is true do something special, otherwise don't. Perl's `if` construct allows us to do exactly that.

The `if` construct looks like this:

```
if (conditional statement) {
        BLOCK
} elsif (conditional statement) {
        BLOCK
} else {
        BLOCK
}
```

Both the `elsif` and `else` parts of the above are optional, and of course you can have more than one `elsif`. Note that `elsif` is also spelled differently to other languages' equivalents --- C programmers should take especial note to not use `else if`.

The parentheses around the conditional are mandatory, as are the curly braces. Perl does not allow dangling statements as does C.

If you're testing whether something is false, you can use the logically opposite construct, `unless`.

```
unless (conditional statement) {
        BLOCK
}
```

The `unless` construct is identical to using `if( not $condition )`. This may be best illustrated by use of an example:

```
# make sure we have apples
if( not $I_have_apples ) {
        go_buy_apples();
}

# now that we have apples...
make_apple_pie();
```

```
# make sure we have apples
unless( $I_have_apples ) {
        go_buy_apples();
}

# now that we have apples...
make_apple_pie();
```

There is no such thing as an `elsunless` (thank goodness!), and if you find yourself using an `else` with `unless` then you should probably have written it as an `if` test in the first place.

There's also a shorthand, and more English-like, way to use `if` and `unless`:

```
print "We have apples\n" if $apples;
print "We have no bananas\n" unless $bananas;
```

So what is a BLOCK?

A block is a hunk of code within curly braces or a file. Blocks can be nested inside larger blocks. The simplest (useful) block is a single statement, for instance:

```
{
        print "Hello, world!\n";
}
```

Sometimes you'll want several statements to be grouped together logically so you can enclose them in a block. A block can be executed either in response to some condition being met (such as after an `if` statement), or as an independent chunk of code that may be given a name.

Blocks always have curly brackets (`{` and `}`) around them. In C and Java, curly brackets are optional in some cases - not so in Perl. Note that it's perfectly acceptable to have a block that is not part a condition or subroutine (called a naked block). We'll see a use for such blocks in our section on *scope*.

```
{
        $fruit = "apple";
        $howmany = 32;
        print "I'd like to buy $howmany ${fruit}s\n";
}
```

You'll notice that the body of the block is indented from the brackets; this is to improve readability. Make a habit of doing it. You'll also recognise our use of `${fruit}` from our discussion on variable interpolation earlier.

The Camel book refers to blocks with curly braces around them as BLOCKs (in capitals). It discusses them on page 111 onwards (97 onwards, 2nd Ed).

Exercises

1. Write `if` statements to test whether the following values are true or false, print an appropriate message for each situation.

   `"00", "", 'false'`

2. Create an array with a number of elements in it. Using an `if` statement test if the entire array is true or false.
3. Repeat the above exercises using `unless`.

Scope

Something that needs mentioning again at this point is the concept of variable scoping. You will recall that we use the `my` function to declare variables when we're using the `strict` pragma. The `my` also scopes the variables so that they are local to the *current block*, which means that these variables are *only* visible inside that block.

```
use strict;
my $a = "foo";
{                               # start a new block
        my $a = "bar";          # a new $a
        print "$a\n";           # prints bar
}
print $a;                       # prints foo
```

We say that the `$a` of the inside block *shadows* the `$a` in the outside block. This is true of all blocks:

```
my $a = 0;
my $b = 0;

unless($a) {
        $a = 5;             # changes $a
        my $b = 5;          # shadows $b (a new $b)
        print "$a, $b\n";   # prints "5, 5"
}
print "$a, $b\n";           # prints "5, 0"
```

Temporary changes to Perl's special variables can be performed by using `local`. It's not possible to use `local` on a lexical variable declared with `my`.

```
$_ = "fish and chips and vinegar";
print $_;               # prints "fish and chips and vinegar"
{
        local $_ = $_;  # allows changes to $_ which only affect this block

        $_ .= " and a pepper pot ";

        print $_;       # prints "fish and chips and vinegar and a pepper pot"
}
                        # $_ reverts back to previous version
print $_;               # prints "fish and chips and vinegar"
```

`local`ising and then changing our variables changes their value not only for the the block we've `local`ised them within, but for every function and subroutine that is called from within that block. As this may not be what you want, it is good practice to keep the scope of our localised variable as small as possible.

Comparison operators

We can compare things, and find out whether our comparison statement is true or not. The operators we use for this are:

Table 8-1. Numerical comparison operators

Operator	Example	Meaning
==	`$a == $b`	Equality (same as in C and other C-like languages)
!=	`$a != $b`	Inequality (again, C-like)
<	`$a < $b`	Less than
>	`$a > $b`	Greater than
<=	`$a <= $b`	Less than or equal to
>=	`$a >= $b`	Greater than or equal to
<=>	`$a <=> $b`	Star-ship operator, see below

The final numerical comparison operator (commonly called the starship operator as it looks somewhat like an ASCII starship) returns -1, 0, or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument. This is commonly seen in use with sorting functions.

If we're comparing strings, we use a slightly different set of comparison operators, as follows:

Table 8-2. String comparison operators

Operator	Example	Meaning
eq	`$a eq $b`	Equality
ne	`$a ne $b`	Inequality

Operator	Example	Meaning
`lt`	`$a lt $b`	Less than (in "asciibetical" order)
`gt`	`$a gt $b`	Greater than
`le`	`$a le $b`	Less than or equal to
`ge`	`$a ge $b`	Greater than or equal to
`cmp`	`$a cmp $b`	String equivalent of <=>

Some examples:

```
69 > 42;                        # true
"0" == 3 - 3;                   # true
'apple' gt 'banana';            # false - apple is asciibetically before
                                #         banana
1 + 2 == "3com";                # true  - 3com is evaluated in numeric
                                #         context because we used == not eq [**]
0 == "fred";                    # true  - fred in a numeric context is 0 [**]
0 eq "fred";                    # false
0 eq 00;                        # true  - both are "0" in string context.
0 eq "00";                      # false - string comparison.  "0" and
                                #         "00" are different strings.
undef;                          # false - undefined is always false.
```

The examples above marked with `[**]` will behave as described but give the following warnings if you use the `-w` flag:

```
Argument "3com" isn't numeric in numeric eq (==) at conditions.pl line 5.
Argument "fred" isn't numeric in numeric eq (==) at conditions.pl line 7.
```

This occurs because although Perl is happy to attempt to massage your data into something appropriate for the expression, the fact that it needs to do so may indicate an error.

Assigning `undef` to a variable name undefines it again, as does using the `undef` function with the variable's name as its argument.

```
my $foo = "bar";
$foo = undef;             # makes $foo undefined
undef($foo);              # same as above
```

Exercises

1. Write a program which takes a value from the command line and compares it using an `if` statement as follows:

   a. If the number is less than 3, print "Too small"

   b. If the number is greater than 7, print "Too big"

   c. Otherwise, print "Just right"

   See `exercises/answers/if.pl` for an answer.

2. Set two variables to your first and last names. Use an `if` statement to print out whichever of them comes first in the alphabet (answer in `exercises/answers/comp_names.pl`).

Existence and definitiveness

We can also check whether things are defined (something is defined when it has had a value assigned to it), or whether an element of a hash exists.

To find out if something is defined, use Perl's `defined` function. The defined function is necessary to distinguish between a value that is false because it is undefined and a value that is false but defined, such as `0` (zero) or `""` (the empty string).

```
my $skippy;                     # $skippy is undefined.
$skippy = "bush kangaroo";      # $skippy is now defined.

if (defined($skippy)) {
        print "Skippy is defined.\n";   # this will be printed.
} else {
        print "Skippy is undefined.\n";
}

$skippy = "";                   # $skippy is still defined but now false.

if($skippy) {                   # This tests for truth, not whether it is
                                # defined.
        print "Skippy is true.\n";
} else {
        print "Skippy is not true\n";   # this will be printed.
}
$skippy = undef;                # $skippy is undefined again.
```

The `defined` function is described in the Camel book on page 697 (page 155, 2nd Ed), and also by **perldoc -f defined**.

To find out if an element of a hash exists, use the `exists` function:

```
my %animals = (
        "Skippy"        =>      "bush kangaroo",
        "Flipper"       =>      "faster than lighting",
        "Blinky Bill"   =>      undef,
);

if (exists($animals{"Blinky Bill"})) {
        print "Blinky Bill exists.\n";         # this will be printed.
} else {
        print "Blinky Bill doesn't exist.\n";
}
```

It's possible for a hash to have an element that is associated with an undefined value. In this case the element `exists` but is not `defined`.

One last quick example to clarify existence, definitives and truth:

```
my %miscellany = (
        "apple"         =>      "red",          # exists, defined, true
        "howmany"       =>      0,              # exists, defined, false
        "name"          =>      "",             # exists, defined, false
        "koala"         =>      undef,          # exists, undefined, false
);
```

Exercise

The following exercise uses the hash below:

```
my %friends = (
        bob     => 'likes television',
        jane    => undef,
        jack    => 0,
        andrew  => "",
        paul    => 'likes perl',
);
```

You can find this hash in `exercises/friends.pl`.

1. Write a program which takes the name of a friend on the command line and returns the fact about that friend. If the friend does not exist in the hash print a note to that effect. Make sure you handle false but defined values reasonably.

   An answer can be found in `exercises/answers/exists.pl`

Boolean logic operators

Boolean logic operators can be used to combine two or more Perl statements, either in a conditional test or elsewhere.

These operators come in two flavours: line noise, and English. Both do similar things but have different precedence. This sometimes causes great confusion. If in doubt, use parentheses to force evaluation order.

Alright, if you insist: `and` and `or` operators have very low precedence (i.e. they will be evaluated after all the other operators in the condition) whereas `&&` and `||` have quite high precedence and may require parentheses in the condition to make it clear which parts of the statement are to be evaluated first.

Table 8-3. Boolean logic operators

English-like	C-style	Example	Result
`and`	`&&`	`$a && $b` `$a and $b`	True if both `$a` and `$b` are true; acts on `$a` then if `$a` is true, goes on to act on `$b`.
`or`	`||`	`$a || $b` `$a or $b`	True if either of `$a` and `$b` are true; acts on `$a` then if `$a` is false, goes on to act on `$b`.
`not`	`!`	`! $a` `not $a`	True if `$a` is false. False if `$a` is true.

Here's how you can use them to combine conditions in a test:

```
my $a = 1;
my $b = 2;

! $a                                    # False
not $a                                  # False
$a == 1 and $b == 2                     # True
$a == 1 or $b == 5                      # True
$a == 2 or $b == 5                      # False
($a == 1 and $b == 5) or $b == 2        # True (parenthesised expression
                                        # evaluated first)
```

Using boolean logic operators as short circuit operators

These operators aren't just for combining tests in conditional statements --- they can be used to combine other statements as well. An example of this is short circuit operations. When Perl sees a true value to the left of a logical `or` operator (either `||` or `or`) it *short circuits* by not evaluating the right hand side, because the statement is already true. When Perl sees a false value to the left of a logical `and` operator it short circuits by not evaluating the right hand side, because the statement is already false. This is fantastic because it means we can put things on the right hand side of these operators that we only want to be executed under certain conditions.

Here's a real, working example of the `||` short circuit operator:

```
open(INFILE, "< input.txt") || die("Can't open input file: $!");
               # Or, more readably:
open(INFILE, "< input.txt") or die("Can't open input file: $!");
```

The `open()` function can be found on page 747 (page 191, 2nd Ed) of the Camel book, if you want to look at how it works. It's also described in **perldoc -f open**.

The `die()` function can be found on page 700 (page 157, 2nd Ed) of the Camel book. Also see **perldoc -f die**.

The `&&` operator is less commonly used outside of conditional tests, but is still very useful. Its meaning is this: if the first operand returns true, the second will also happen. As soon as you get a false value returned, the expression stops evaluating.

```
($day eq 'Friday') && print "Have a good weekend!\n";
```

The typing saved by the above example is not necessarily worth the loss in readability, especially as it could also have been written:

```
print "Have a good weekend!\n" if $day eq 'Friday';

if ($day eq 'Friday') {
        print "Have a good weekend!\n";
}
```

...or any of a dozen other ways. That's right, there's more than one way to do it.

The most common usage of the short circuit operators, especially `||` (or `or`) is to trap and handle errors, such as when opening files or interacting with the operating system.

 Short circuit operators are covered from page 102 (page 89, 2nd Ed) of the Camel book.

Boolean assignment

Boolean logic operators are great in conditional statements and as short circuit operators but they can also be used in assignment. Consider the following:

```
$a ||= 0;         # Which is short hand for $a = $a || 0;
```

What does this do for us? It says, if `$a` is not true (that is; if `$a` is any of `0` (zero), `""` (the empty string) or `undef` (the undefined value)), set it to be `0`. After this statement we know for certain that `$a` is both defined and valid, even if it isn't true.

Loop conditional constructs

Often when coding you wish to do the same task a number of times. For example for every line of a file, or while there is input from the user or for every element of an array. This is why there are looping constructs.

while loops

We can repeat a block while a given condition is true:

```
while (conditional statement) {
        BLOCK
}

        # if $hunger <= 0 to start with, this will never start.
my $hunger = 5;
while ($hunger > 0) {
        print "Feed me!\n";
        $hunger--;
}
```

The logical opposite of this is the "until" construct:

```
$full = -5;
until ($full > 0) {
        print "Feed me!\n";
        $full++;
}
```

Like the `if` and `unless` constructs, `while` and `until` also have their shorthand forms:

```
print "Feed me!\n" while ($hunger-- > 0);
print "Feed me!\n" until ($full++ > 0);
```

Like the shorthand conditionals, these forms may only have a single statement on the left hand side.

do while loops

Those familiar with C may wonder whether Perl also has `do while` loops as well. The answer is that it certainly does. `do while` loops are great if you want make sure that the contents of a loop execute the first time, even if the condition is false.

`do while` loops look like the following:

```
my @titles = ("The Hobbit", "Fellowship of the Ring", "The Two Towers",
              "Return of the King", "Great Expectations");
my $title = "*** I have these books ***";

do {
       print $title, "\n";
} while ($title = shift @titles);
```

`do until` loops also exist.

for and foreach

Perl has a `for` construct identical to C and Java:

```
for (my $count = 0; $count <= $enough; $count++) {
        print "Had enough?\n";
}
```

However, since we often want to loop through the elements of an array, we have a special "shortcut" looping construct called `foreach`, which is similar to the construct available in some Unix shells. Compare the following:

```
# using a for loop

for (my $i = 0; $i < @array; $i++) {
        print $array[$i] . "\n";
}

# using foreach

foreach (@array) {
        print "$_\n";
}
```

You'll notice above that we used the special variable `$_` to print each element in our array. `foreach` doesn't have to bind `$_` to the array elements, you can name your own variable to use instead.

```
foreach my $value (@array) {
        print "$value\n";
}
```

Naming the variable you want to bind with in `foreach` is often good programming practice, as it can make your code much more readable. However, there are cases when allowing Perl to bind to `$_` results in better looking code. We'll see some examples of this later on when we cover regular expressions.

There are some examples of foreach in `exercises/foreach.pl`

When in a `foreach` loop, the variable representing the current element *is* the current element, not just a copy of it. This means that if you change this variable, you change the original. For example, the following loop will double all the numbers in a list:

```
foreach (@numbers) {
        $_  = $_ * 2;
}
```

`foreach(n..m)` can be used to automatically generate a list of numbers between `n` and `m`.

We can loop through hashes easily too, using the `keys` function to return the keys of a hash as an list that we can use:

```
foreach my $month (keys %monthdays) {
        print "There are $monthdays{$month} days in $month.\n";
}
```

`foreach` constructs may also be used in a trailing form:

```
print $_ foreach (@array);
```

Exercises

1. Create a hash and print out the keys and values for each item using a foreach loop (hint: look up the `keys` function in your Camel book or use **perldoc -f keys**). A starter can be found in `exercises/loops_starter.pl`
2. Use a `while` loop to print out a numbered list of the elements in an array
3. Now do it with a `for` loop
4. Try it with a `foreach` loop (this is a little harder).

Answers are given in `exercises/answers/loops.pl`

Practical uses of `while` loops: taking input from STDIN

STDIN is the standard input stream for any Unix program. If a program is interactive, it will take input from the user via STDIN. Many Unix programs accept input from STDIN via pipes and redirection. For instance, the Unix **cat** utility prints out all the files given to it on the command line, but will also print out files redirected to its STDIN:

```
% cat < hello.pl
```

Unix also has STDOUT (the standard output) and STDERR (where errors are printed to).

We can get a Perl script to take input from STDIN (standard input) and do things with it by using the line input operator, which is a set of angle brackets with the name of a filehandle in between them:

```
my $user_input = <STDIN>;
```

The above example takes a single line of input from STDIN. The input is terminated by the user hitting Enter. If we want to repeatedly take input from STDIN, we can use the line input operator in a `while` loop:

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

while ($_ = <STDIN>) {
        # do some stuff here, if you want...
        print;     # remember that print takes $_ as its default argument
}
```

When Perl sees a simple assignment from a filehandle as the condition of a while loop, it automatically checks that the value is defined rather that just true. This saves us from having to write:

```
while (defined($_ = <STDIN>)) {
```

which we'd otherwise have to do.

Input continues to be taken until the end of file character (commonly written EOF) is encountered. To end user input and terminate this loop press **CTRL-D** (a.k.a. `^D`) to indicate end of file.

Conveniently enough, the `while` statement can be written more succinctly, because inside a `while` or `until` loop, the line input operator assigns to `$_` by default:

```
while (<STDIN>) {
        print;
}
```

The above construct is exactly equivalent to our previous example.

The readline operator also has its own default behaviour, so in most circumstances we can shorten the above loop even further:

```
while (<>) {
        print;
}
```

The <> (diamond) construct is highly magical. It opens and reads files listed on the command line (from `@ARGV`), or from `STDIN` if no files are listed. This is an incredible useful construct that is well worth remembering.

As always, there's more than one way to do it.

Exercises

The above example script (which is available in your directory as `exercises/cat.pl`) will basically perform the same function as the Unix **cat** command; that is, print out whatever's given to it through STDIN.

1. Try running the script with no arguments. You'll have to type some stuff in, line by line, and type **CTRL-D** (a.k.a. `^D`) when you're ready to stop. `^D` indicates end-of-file (EOF) on most Unix systems.
2. Now try giving it a file by using the shell to redirect its own source code to it:

   ```
   perl exercises/cat.pl < exercises/cat.pl
   ```

   This should make it print out its own source code.

3. Since the `cat.pl` program uses the diamond construct, it will also process files presented on the command line. Use it to display the concatenated contents of a couple of other files.

Named blocks

Blocks can be given names, thus:

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

LINE: while (<STDIN>) {
        ...
}
```

By tradition, the names of blocks are in upper case. The name should also reflect the type of things you are iterating over --- in this case, lines of text from STDIN.

Breaking out or restarting loops

You can change loop flow (to restart or end) by using the functions `next`, `last` and `redo`.

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

LINE: while (<STDIN>) {
        chomp;                          # remove newline
        next LINE if $_ eq "";          # skip blank lines
        last LINE if lc($_) eq 'q';     # quit
}
```

Writing `next LINE` tells Perl to repeat the block named `LINE` from the start. Writing `last LINE` tells Perl to jump to the end of the block named `LINE` and continue program execution. By default `next` and `last` affect the current smallest loop. In the example above the current smallest loop is the while loop so the block name `LINE` could have been omitted leaving us with:

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

while (<STDIN>) {
        chomp;                  # remove newline
        next if $_ eq "";       # skip blank lines
        last if lc($_) eq 'q';  # quit
}
```

Named blocks are most useful when we wish to break out of a loop higher up the chain:

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

LINE: while (<STDIN>) {
        chomp;                                  # remove newline
        next if $_ eq "";                       # skip blank lines

        # we split the line into words and check all of them
        foreach (split) {
                last LINE if lc($_) eq 'quit';  # quit
        }
}
```

There is another loop flow function named `redo`. `redo` allows you to restart the loop at the top without evaluating the conditional again. This command is not used very often but it useful for programs which want to lie to themselves about what they've just seen. For example:

```
while(<>) {
        if(/\d{4}-\d{2}-\d{2}/) {       # it looks like it's a date line
                # do something long and complicated
        }
        elsif(\w+\.\w+\.\w) {           # it looks like it's a host line
                # do something else long and complicated
        }
        elsif(\.html?) {                # looks like it's a page line
                # another long and complicated thing
        }
        else {                          # it doesn't look right at all
                                        # perhaps it's encoded...
                $_ = unpack('u', $_);
                if($_) {                # looks like it was uuencoded
                        redo;           # try unencoded string
                }
                else {
                        print "Unexpected input: $_\n";
                }
        }
}
```

In this case, had we used `next` our changes to `$_` would have been lost when we re-evaluated the conditional `<>`. `redo` refers to the innermost enclosing loop by default but can also take a LABEL like `next` and `last`.


Checkout **perldoc -f last**, **perldoc -f next** and **perldoc -f redo** for information on these functions.

Practical exercise

Write a program which generates a random integer between 1 and 100 and then asks the user to guess it. Verify that the user's guess is a number between 1 and 100 before proceeding. If it is not, tell the user and ask for a new number. If the user guesses too high, or too low, tell them so and ask again. If the user gets the number correct; terminate the loop and congratulate them. Count how many guesses it required and report this at the end.

An answer can be found in `exercises/answers/guessing_game.pl`. Try the exercise first before looking at the answer.

Chapter summary

- A block in Perl is a series of statements grouped together by curly braces. Blocks can be used in conditional constructs and subroutines.
- A conditional construct is one which executes statements based on the truth of a condition.

- Truth in Perl is determined by testing whether something is NOT any of: numeric zero, the empty string, or undefined.
- The `if - elsif - else` conditional construct can be used to perform certain actions based on the truth of a condition.
- The `unless` conditional construct is equivalent to `if(not(...))`.
- The `while`, `for`, and `foreach` constructs can be used to repeat certain statements based on the truth of a condition.
- A common practical use of the `while` loop is to read each line of a file.
- Blocks may be named using the `NAME:` convention.
- You can change execution flow in blocks by using `next`, `redo` and `last`.

Chapter 9. Subroutines

In this chapter...

In this chapter, we look at subroutines and how they can be used to simplify your code. More advanced material regarding subroutines and parameter passing can be found in Appendix B.

Introducing subroutines

If you have a long Perl script, you'll probably find that there are parts of the script that you want to break out into subroutines (sometimes called functions in other languages). In particular, if you have a section of code which is repeated more than once, it's best to make it a subroutine to save on maintenance (and, of course, line count).

What is a subroutine?

A subroutine is a set of statements which performs a specific task.

The statements in a subroutine are compiled into a unit which can then be called from anywhere in the program. This allows your program to access the subroutine repeatedly, without the subroutine's code having been written more than once.

Subroutines are very much like Perl's functions, however you can define your own subroutines for the tasks at hand. We use Perl's `print` function to output data, rather than writing output code for each and every character we wish to output. In a similar way, we can write subroutines to allow us to reuse the same block of code from different parts of our program.

Just like Perl's `print` function can take arguments, so can your subroutines. They can also return values of any type. If you find yourself repeating a task, it's often best to consider what it needs to do its task, and what information it needs to return, and then write your code into a subroutine.

Why use subroutines?

By creating your own subroutines, you are able to reduce code repetition and improve code maintainability. For example; rather than writing code to send an email to the administrator, and a separate block of code to send an email to a user; you could combine the email sending code into a single subroutine. Once written, you can then call this subroutine with the recipient of the mail and what you wish to send them. Now if you ever need to change how an e-mail is sent, you only need to change your code in one location.

Subroutines are used:

- to avoid or reduce redundant code,
- to improve maintainability and reduce possibility of errors,
- to reduce complexity by breaking complex problems into smaller, more simple pieces,
- to improve readability in the program,

Using subroutines in Perl

A subroutine is basically a little self-contained mini-program in the form of block which has a name, and can take arguments and return values:

```
# the general case

sub name {
        BLOCK
}

# a specific case

sub print_headers {
        print "Programming Perl, 2nd ed\n";
        print "by\n";
        print "Larry Wall et al.\n";
}
```

Perl subroutines don't come with declarations as they do in C and some other languages. This means that (usually) you cannot rely on the compiler to verify that you have passed your in your arguments in the correct order, and to ensure that you haven't missed any. This is an advantage if you wish to be able to call your subroutine and leave off optional arguments, but it can be surprising at first.

Calling a subroutine

A subroutine can be called in any of the following ways:

```
print_headers();                # The preferred method.
&print_headers();               # Sometimes necessary.
&print_headers;                 # An older style (with some dangers).
print_headers;                  # Ambiguous, can cause problems under strict.
```

If (for some reason) you've got a subroutine that clashes with one of Perl's functions you will need to prefix your function name with `&` (ampersand) to be perfectly clear. For example: `&sort(@array)` if you have your own `sort` function, but don't do that. You should avoid naming your functions after Perl's built-in functions because it typically causes more confusion than it's worth. Especially to whichever poor soul tries to maintain your code.

Be careful of calling your functions in the form `&print_headers;` as this can result in a rather surprising effect. For historical reasons, calling your subroutines prepended with an ampersand and excluding arguments means that the subroutine is passed with an implicit argument list, which is everything currently in `@_`. While, occasionally, this may be intentional, writing `print_headers(@_)` will make your code much easier for other people to understand.

There are other times when you need to use an ampersand on your subroutine name, such as when a function needs a SUBROUTINE type of parameter, or when making an anonymous subroutine reference.

Passing arguments to a subroutine

You can pass arguments to a subroutine by including them in the parentheses when you call it. The arguments end up in a special array called `@_` which is only visible inside the subroutine.

Passing in scalars

The most common variable type passed into a subroutine is the scalar.

```
print_headers("Programming Perl, 2nd ed", "Larry Wall et al");

my $fiction_title = "Lord of the Rings";
my $fiction_author = "J.R.R. Tolkein";

print_headers($fiction_title, $fiction_author);

sub print_headers {
        my ($title, $author) = @_;
        print "$title\n";
        print "by\n";
        print "$author\n";
}
```

You can take any number of scalars in as arguments - they'll all end up in `@_` in the same order you gave them.

Inside a subroutine, the `shift` function will by default shift and return arguments from the start of `@_`. As such, it's also very common to see code like this:

```
sub print_headers {
        my $title  = shift || "Untitled";
        my $author = shift || "Anonymous";
        print "$title\n
        print "by\n
        print "$author\n";
}
```

One use of this is when you pass a different number of arguments to a function depending on what you want it to do. Try to avoid `shift`ing arguments from `@_` deep down into your subroutine. Doing this will make it much harder for someone to maintain your code later.

Passing in arrays and hashes

To pass in a single array or hash to a subroutine, make it the final element in your argument list. For example:

```
print_headers($title, $author, @publication_dates);

sub print_headers {
        my ($title, $author, @dates) = @_;
        print "$title\nby\n$author\n";
        if(@dates) {    # If we were given any publication dates
                print "Printed on: @dates";
        }
}
```

Passing in more than one array or hash causes problems. This is because of list flattening. When Perl sees a number of items in parentheses these are combined into one big list.

```
# Flatten two lists into one big list and put that in an array
my @biglist = (@list1, @list2);

# Flatten (join) two hashes into one big list an put that in a hash
my %bighash = (%hash1, %hash2);

# Make a nonsense list and put that in an array:
my @nonsense = (%bighash, @list1, @biglist, 1 .. 4);
```

Thus if we write the following code, we won't get the results we want:

```
my @colours = qw/red blue white green pink/;
my @chosen = qw/red white green/;

print_unchosen(@chosen, @colours);

sub print_unchosen {
        my (@chosen, @colours) = @_;

                # at this point @chosen contains:
                # (red white green red blue white green pink)
                # and @colours contains () - the empty list.
}
```

Once lists have been flattened, Perl is unable to tell where one list stopped and the other started. Thus when we attempt to separate `@chosen` and `@colours` into their original lists, `@chosen` takes all the elements and leaves `@colours` empty. This will happen with hashes too.


We can avoid this problem by using references:

```
my @colours = qw/red blue white green pink/;
my @chosen = qw/red white green/;

print_unchosen(\@chosen, \@colours);

sub print_unchosen {
        my ($chosen, $colours) = @_;

        my @chosen = @$chosen;
        my @colours = @$colours;

                # at this point @chosen contains:
                # (red white green)
                # and @colours contains (red blue white green pink)
}
```

however these are beyond the scope of this section.


To learn more about references read **perldoc perlreftut**.

Returning values from a subroutine

To return a value from a subroutine, simply use the `return` function.

```
sub format_headers {
        my ($title, $author) = @_;
        return "$title\nby\n$author\n\n";
}

sub sum {
        my $total = 0;
        foreach (@_) {
                $total = $total + $_;
        }
        return $total;
}
```

These return values could be used as follows:

```
my $header = format_headers("War and Peace", "Leo Tolstoy");
print $header;

my $total = sum(1..100);
print "$total\n";

my $silly_total = sum($total, length($header));
print "$silly_total\n";
```

You can also return lists from your subroutine:

```
# subroutine to return the first three arguments passed to it
sub firstthree {
        return @_[0..2];
}

my @three_items = firstthree("x", "y", "z", "a", "b");
# sets @three_items to ("x", "y", "z");

# alternately:
my ($x, $y, $z) = firstthree(4..10);  # set $x = 4, $y = 5, $z = 6
```

Occasionally you might want to return different information based on the context in which your subroutine was called. For example `localtime` returns a human-readable time string when called in scalar context and a list of time information in list context.

To achieve this you can use the `wantarray` function. To learn more about this, read pg 827 in the Camel book (pg 241, 2nd Ed) and **perldoc wantarray**.

Exercises

1. Write a subroutine which prints out its first argument.
2. Modify the above subroutine to also print out the last argument.

3. Now change it to compare the first and last arguments and return the one which is asciibetically larger (you'll want to use an `if` statement for that).

You'll find the answers the the above in `exercises/answers/subroutines.pl`

Chapter summary

- A subroutine is a named block which can be called from anywhere in your Perl program.
- Subroutines can accept parameters, which are available via the special array `@_`.
- Arrays and hashes should be passed as the last argument to subroutines. In the case where it is necessary to pass more than one array or hash to a subroutine references must be used.
- Subroutines can return scalar or list values.

Chapter 10. Regular expressions

In this chapter...

In this chapter we begin to explore Perl's powerful regular expression capabilities, and use regular expressions to perform matching and substitution operations on text.

Regular expressions are a big reason of why so many people learn Perl. In fact Perl's regular expressions have lead the field and been copied by many other popular languages such as Java, PHP, Python and Ruby. One of Perl's most common uses is string processing and it excels at that because of its support for regular expressions.

 Patterns and regular expressions are dealt with in depth in chapter 5 (chapter 2, 2nd Ed) of the Camel book, and further information is available in the online Perl documentation by typing **perldoc perlre**.

What are regular expressions?

The easiest way to explain this is by analogy. You will probably be familiar with the concept of matching filenames under DOS and Unix by using wild cards - `*.txt` or `/usr/local/*` for instance. When matching filenames, an asterisk can be used to match any number of unknown characters, and a question mark matches any single character. There are also less well-known filename matching characters.

Regular expressions are similar in that they use special characters to match text. The differences are that more powerful text-matching is possible, and that the set of special characters is different.

Regular expressions are also known as REs, regexes, and regexps.

Regular expression operators and functions

m/PATTERN/ - the match operator

The most basic regular expression operator is the matching operator, `m/PATTERN/`.

- Works on `$_` by default.
- In scalar context, returns true (`1`) if the match succeeds, or false (the empty string) if the match fails.
- In list context, returns a list of any parts of the pattern which are enclosed in parentheses. If there are no parentheses, the entire pattern is treated as if it were parenthesised.
- The `m` is optional if you use slashes as the pattern delimiters.
- If you use the `m` you can use any delimiter you like instead of the slashes. This is very handy for matching on strings which contain slashes, for instance directory names or URLs.

- Using the `/i` modifier on the end makes it case insensitive.

```
while (<>) {
        print if m/foo/;        # prints if a line contains "foo"
        print if m/foo/i;       # prints if a line contains "foo", "FOO", etc
        print if /foo/i;        # exactly the same; the m is optional
        print if m#foo#i;       # the same again, using different delimiters
        print if /http:\/\//;   # prints if a line contains "http://"
                                # suffers from "leaning-toothpick-syndrome".
        print if m!http://!;    # using ! as an alternative delimiter
        print if m{http://};    # using {} as delimiters
}
```

s/PATTERN/REPLACEMENT/ - the substitution operator

This is the substitution operator, and can be used to find text which matches a pattern and replace it with something else.

- Works on `$_` by default.
- In scalar context, returns the number of matches found and replaced.
- In list context, behaves the same as in scalar context and returns the number of matches found and replaced (a cause of more than one mistake...).
- You can use any delimiter you want, the same as the `m//` operator.
- Using `/g` on the end of it matches globally, otherwise matches (and replaces) only the first instance of the pattern.
- Using the `/i` modifier makes it case insensitive.

```
# fix some misspelled text

while (<>) {
        s/freind/friend/g;    # Correct freind to friend on entire line.
        s/teh/the/g;
        s/jsut/just/g;
        s/pual/Paul/ig;       # Correct (case insensitive) all occurrences
                              # of "pual" (or "Pual" or "PuAl" etc)
        print;
}
```

Exercises

The above example can be found in `exercises/spellcheck.pl`.

1. Run the spelling check script over the `exercises/spellcheck.txt` file.
2. There are a few spelling errors remaining. Change your program to handle them as well.

Binding operators

If we want to use `m//` or `s///` to operate on something other than `$_` we need to use binding operators to bind the match to another string.

Table 10-1. Binding operators

Operator	Meaning
`=~`	True if the pattern matches
`!~`	True if the pattern doesn't match

```
print "Please enter your homepage URL: ";
my $url = <STDIN>;

if($url !~ /^http:/) {
        print "Doesn't look like a http URL.\n";
}

if ($url =~ /geocities/) {
        print "Ahhh, I see you have a geocities homepage!\n";
}

my $string = "The act sat on the mta";
$string =~ s/act/cat/;
$string =~ s/mta/mat/;

print $string;  # prints: "The cat sat on the mat";
```

Easy Modifiers

There are several modifiers for regular expressions. We've seen two already.

Table 10-2. Regexp modifiers

Modifier	Meaning
`/i`	Make match/substitute match case insensitive
`/g`	Make substitute global (all occurrences are changed)


You can find out about the other modifiers by reading **perldoc perlre**.

Meta characters

The special characters we use in regular expressions are called *meta characters*, because they are characters that describe other characters.

Some easy meta characters

Table 10-3. Regular expression meta characters

Meta character(s)	Matches...
`^`	Start of string
`$`	End of string
`.`	Any single character except `\n`
`\n`	Newline
`\t`	Matches a tab
`\s`	Any whitespace character, such as space, tab, or newline
`\S`	Any non-whitespace character
`\d`	Any digit (0 to 9)
`\D`	Any non-digit
`\w`	Any "word" character - alphanumeric plus underscore (_)
`\W`	Any non-word character
`\b`	A word break - the zero-length point between a word character (as defined above) and a non-word character.
`\B`	A non-word break - anything other than a word break.


These and other meta characters are all outlined in chapter 5 (chapter 2, 2nd Ed) of the Camel book and in the `perlre` manpage - type **perldoc perlre** to read it.


It's possible to use the `/m` and `/s` modifiers to change the behaviour of the first three meta characters (`^`, `$`, and `.`) in the table above. These modifiers are covered in more detail later in the course.


Under newer versions of Perl, the definitions of spaces, words, and other characters is locale-dependent. Usually Perl ignores the current locale unless you ask it to do otherwise, so if you don't know what's meant by locale, then don't worry.

Any character that isn't a meta character just matches itself. If you want to match a character that's normally a meta character, you can escape it by preceding it with a backslash.

Some quick examples:

```
# Perl regular expressions are often found within slashes

/cat/                                   # matches the three characters
                                        # c, a, and t in that order.
```

```
/^cat/                                  # matches c, a, t at start of line

/\scat\s/                               # matches c, a, t with spaces on
                                        # either side

/\bcat\b/                               # Same as above, but won't
                                        # include the spaces in the text
                                        # it matches.  Also matches if
                                        # cat is at the very start or
                                        # very end of a string.

# we can interpolate variables just like in strings:

my $animal = "dog"                      # we set up a scalar variable
/$animal/                               # matches d, o, g
/$animal$/                              # matches d, o, g at end of line

/\$\d\.\d\d/                            # matches a dollar sign, then a
                                        # digit, then a dot, then
                                        # another digit, then another
                                        # digit, eg $9.99
                                        # Careful! Also matches $9.9999
```

Quantifiers

What if, in our last example, we'd wanted to say "Match a dollar, then any number of digits, then a dot, then only two more digits"? What we need are quantifiers.

Table 10-4. Regular expression quantifiers

Quantifier	Meaning
?	0 or 1
*	0 or more
+	1 or more
{n}	match exactly n times
{n,}	match n or more times
{n,m}	match between n and m times

Here are some examples to show you how they all work:

```
/Mr\.? Fenwick/;            # Matches "Mr. Fenwick" or "Mr Fenwick"
/camel.*perl/;              # Matches "camel" before "perl" in the
                            # same line.
/\w+/;                      # One or more word characters.
/x{1,10}/;                  # 1-10 occurrences of the letter "x".
```

Exercises

For these exercises you may find using the following structure useful:

```
while(<>) {
        chomp;

        print "$_ matches!\n" if (/PATTERN/);   # put your regexp here
}
```

This will allow you to specify test files on the command line to check against, or to provide input via STDIN. Hit **CTRL-D** to finish entering input via STDIN. (Use the key combination **CTRL-Z** on Windows).

You can find the above snippet in: `exercises/regexploop.pl`.

1. Earlier we mentioned writing a regular expression for matching a price. Write one which matches a dollar sign, any number of digits, a dot and then exactly two more digits.

   Make sure you're happy with its performance with test cases like the following: `12.34`, `$111.223`, `$.24`.

2. Write a regular expression to match the word "colour" with either British or American spellings (Americans spell it "color")?

3. How can we match any four-letter word?

See `exercises/answers/regexp.pl` for answers.

Grouping techniques

Let's say we want to match any lower case character. `\w` matches both upper case and lower case so it won't do what we need. What we need here is the ability to match any characters in a *group*.

Character classes

A character class can be used to find a single character that matches any one of a given set of characters.

Let's say you're looking for occurrences of the word "grey" in text, then remember that the American spelling is "gray". The way we can do this is by using character classes. Character classes are specified using square brackets, thus: `/gr[ea]y/`

We can also use character sequences by saying things like `[A-Z]` or `[0-9]`. The sequences `\d` and `\w` can easily be expressed as character classes: `[0-9]` and `[a-zA-Z0-9_]` respectively.

Inside a character class some characters take on special meanings. For example, if the first character is a caret, then the list is negated. That means that `[^0-9]` is the same as `\D` --- that is, it matches any non-digit character.

Here are some of the special rules that apply inside character classes.

- `^` at the start of a character class negates the character class, rather than specifying the start of a line.

- `-` specifies a range of characters. If you wish to match a literal -, it must be either the first or the last character in the class.
- `$ . () {} * +` and other meta characters taken literally.

Exercises

Your instructor will help you do the following exercises as a group.

1. How would we find any word starting with a letter in the first half of the alphabet, or with X, Y, or Z?
2. What regular expression could be used for any word that starts with letters *other* than those listed in the previous example.
3. There's almost certainly a problem with the regular expression we've just created - can you see what it might be?

Alternation

The problem with character classes is that they only match one character. What if we wanted to match any of a set of longer strings, like a set of words?

The way we do this is to use the pipe symbol `|` for alternation:

```
/rabbit|chicken|dog/                # matches any of our pets
```


The pipe symbol (also called *vertical bar*) is often found on the same key as `\`.

However this will match a number of things we might not intend it to match. For example:

- rabbiting
- chickenhawk
- hotdog

We need to specify that we want to only match the word if it's on a line by itself.

Now we come up against another problem. If we write something like:

```
/^rabbit|chicken|dog$/
```

to match any of our pets on a line by itself, it won't work quite as we expect. What this actually says is match a string that:

- starts with the string "rabbit" or
- has the string "chicken" in it or
- ends with the string "dog"

This will still match the three incorrect words above, which is not what we intended. To fix this, we enclose our alternation in round brackets:

```
/^(rabbit|chicken|dog)$/
```

Finally, we will now only match any of our pets on a line, by itself.

Alternation can be used for many things including selecting headers from emails for printing out:

```
# a simple matching program to get some email headers and print them out

while (<>) {
        print if /^(From|Subject|Date):\s/;
}
```

The above email example can be found in `exercises/mailhdr.pl`.

The concept of atoms

Round brackets bring us neatly into the concept of atoms. The word "atom" derives from the Greek *atomos* meaning "indivisible" (little did they know!). We use it to mean "something that is a chunk of regular expression in its own right".

Atoms can be arbitrarily created by simply wrapping things in round brackets --- handy for indicating grouping, using quantifiers for the whole group at once, and for indicating which bit(s) of a matching function should be the returned value.

In the example used earlier, there were three atoms:

1. start of line
2. rabbit or chicken or dog
3. end of line

How many atoms were there in our dollar prices example earlier?

Atomic groupings can have quantifiers attached to them. For instance:

```
# match a consonant followed by a vowel twice in a row
# eg "tutu" or "tofu"
/([^aeiou][aeiou]){2}/;

# match three or more words starting with "a" in a row
# eg "all angry animals"
/(\ba\w+\b\s*){3,}/;
```

Exercises

1. Determine whether your name appears in a string (an answer's in `exercises/answers/namere.pl`).
2. Remove footnote references (like [1]) from some text (see `exercises/footnote.txt` for some sample text, and `exercises/answers/footnote.pl` for an answer). (Hint: have a look at the footnote text to determine the forms footnotes can take).
3. What pattern could be used to match a blank line? (Answer: `exercises/answers/blanklinere.pl`)

4. Write a script to search a file for any of the names "Yasser Arafat", "Boris Yeltsin" or "Monica Lewinsky". Print out any lines which contain these names. (Answer: `exercises/answers/namesre.pl`)
5. What pattern could be used to match any of: Elvis Presley, Elvis Aron Presley, Elvis A. Presley, Elvis Aaron Presley. (Answer: `exercises/answers/elvisre.pl`)
6. What pattern could be used to match an IP address such as `192.168.53.124`, where each part of the address is a number from 0 to 255? (Answer: `exercises/answers/ipre.pl`)

Chapter summary

- Regular expressions are used to perform matches and substitutions on strings.
- Regular expressions can include meta-characters (characters with a special meaning, which describe sets of other characters) and quantifiers.
- Character classes can be used to specify any single instance of a set of characters.
- Alternation may be used to specify any of a set of sub-expressions.
- The matching operator is `m/PATTERN/` and acts on `$_` by default.
- The substitution operator is `s/PATTERN/REPLACEMENT/` and acts on `$_` by default.
- Matches and substitutions can be performed on strings other than `$_` by using the `=~` (and `!~`) binding operator.

Chapter 11. References and complex data structures

In this chapter...

In this chapter, we look at Perl's powerful reference syntax and how it can be used to implement complex data structures such as multi-dimensional lists, hashes of hashes, and more.

Assumed knowledge

It is assumed that you have a good understanding of Perl's data types: scalars, arrays, and hashes. Prior experience with languages which use pointers or references is helpful, but not required.

Introduction to references

Perl's basic data type is the *scalar*. Arrays and hashes are made up of scalars, in one- or two-dimensional lists. It is not possible for an array or hash to be a member of another array or hash under normal circumstances.

However, there is one thing about an array or hash which is scalar in nature -- its memory address. This memory address can be used as an item in an array or list, and the data extracted by looking at what's stored at that address. This is what a reference is.

The following sources also provide useful and comprehensive information about references:

- Chapter 8 (chapter 4, 2nd Ed) of the Camel book, and in **perldoc perlref**.
- Chapter 1 of Advanced Perl Programming (O'Reilly's Panther book).
- Tom Christiansen's FMTYEWTK (Far More Than You Ever Wanted To Know) tutorials available from the Perl website (http://www.perl.com/).

Uses for references

There are three main uses for Perl references.

Creating complex data structures

Perl references can be used to create complex data structures, for instance hashes of arrays, arrays of hashes, hashes of hashes, and more.

Passing arrays and hashes to subroutines and functions

Since all arguments to subroutines are flattened to a list of scalars, it is not possible to use two arrays as arguments and have them retain their individual identities.

```
my @a1 = qw(a b c);
my @a2 = qw(d e f);

printargs(@a1, @a2);

sub printargs {
        print "@_\n";
}
```

The above example will print out `a b c d e f`.

References can be used in these circumstances to keep arrays and hashes passed as arguments separate.

Object oriented Perl

References are used extensively in object oriented Perl. In fact, Perl objects *are* references to data structures.

Creating and dereferencing references

To create a reference to a scalar, array or hash, we prefix its name with a backslash:

```
my $scalar = "This is a scalar";
my @array  = qw(a b c);
my %hash = (
        'sky'           =>      'blue',
        'apple'         =>      'red',
        'grass'         =>      'green',
);

my $scalar_ref = \$scalar;
my $array_ref  = \@array;
my $hash_ref   = \%hash;
```

Note that all references are scalars, because they contain a single item of information: the memory address of the actual data. This is what a reference looks like if you print it out:

```
% perl -e 'my $scalar_ref = \$scalar; print "$scalar_ref\n";'
SCALAR(0x80c697c)
% perl -e 'my $array_ref  = \@array;  print "$array_ref\n";'
ARRAY(0x80c6988)
% perl -e 'my $hash_ref   = \%hash;   print "$hash_ref\n";'
HASH(0x80c6988)
```

You can find out whether a scalar is a reference or not by using the `ref()` function, which returns a string indicating the type of reference, or undef if the scalar is not a reference.

```
print ref($scalar_ref);          # prints SCALAR
print ref($array_ref);           # prints ARRAY
print ref($hash_ref);            # prints HASH
```


The `ref()` function is documented on page 773 (page 204, 2nd Ed) of the Camel book or in **perldoc -f ref**.

Dereferencing (getting at the actual data that a reference points to) is achieved by prepending the appropriate sigil to the name of the reference. For instance, if we have a hash reference `$hash_reference` we can dereference it by adding a percentage sign: `%$hash_reference`.

```
my $new_scalar = $$scalar_ref;
my @new_array  = @$array_ref;
my %new_hash   = %$hash_ref;
```

Here's one way to access array elements or slices, and hash elements:

```
print $$array_ref[0];           # prints the first element of the array
                                # referenced by $array_ref: a
print @$array_ref[1,2];         # prints an array slice: b, c
print $$hash_ref{'sky'};        # prints a hash element's value: blue
```

The other way to access the value that a reference points to is to use the "arrow" notation. This notation is usually considered to be better Perl style than the one shown above, which can have precedence problems and is less visually clean.

```
print $array_ref->[0];       # prints the first element of the array
                             # referenced by $array_ref: a
print $hash_ref->{'sky'};    # prints a hash element's value: blue
```

The notation here is exactly the same as selecting elements from an array or hash, except that an arrow is inserted between the variable name and the element to fetch. So where `$foo[1]` gets the first (ie, position 2) element from the array `@foo`, `$foo->[1]` gets the first element from the array pointed to by the reference `$foo`.


It's not possible to get an array or hash slice using arrow notation.

Taking an array slice of a single element from an array reference does not result in a warning from Perl, although it's certainly not recommended. Perl does however try to be helpful in this case and returns the scalar referred to by the array slice, rather than the length of the array slice which would be `1`.

```
my $value = @$array_ref[0];     # Oops, this should be $$array_ref[0];
print $value;                   # Prints 'a' as desired but is not obvious
```

Exercises

1. Create an array called `@friends`, and populate it with the name of some of your friends.
2. Create a reference to your array called `$friends_ref`. Using this reference, print the names of three of your friends.

Assigning through references

Assigning values to the underlying array or hash through a reference is much the same as accessing the value:

```
my @trees = qw/lemon orange grapefruit/;
my $treeref = \@trees;

$treeref->[3] = 'mandarin';
print "@trees";                 # prints "lemon orange grapefruit mandarin"

my %fruit = ( kumquat => "sour",
              orange => "sweet",
              lemon => "sour",
              mandarin => "sweet");

my $fruit = \%fruit;            # scalars, arrays and hashes can all
                                # have the same names if it makes sense

$fruit->{grapefruit} = "sour and sweet";
```

Passing multiple arrays/hashes as arguments

When we pass multiple arrays to a subroutine they are flattened out to form one large array.

```
my @colours = qw/red blue white green pink/;
my @chosen = qw/red white green/;

print_unchosen(@chosen, @colours);

sub print_unchosen {
        my (@chosen, @colours) = @_;

        # at this point @chosen contains:
        # (red white green red blue white green pink)
        # and @colours contains () - the empty list.
}
```

If we want to keep them separate, we need to pass in references to the arrays instead:

```
ref_print_unchosen(\@chosen, \@colours);

sub ref_print_unchosen {
        my ($chosen, $colours) = @_;

        print "Chosen list:\n";
        foreach (@$chosen) {
                print "$_\n";
        }
        print "Colour list:\n";
        foreach (@$colours) {
                print "$_\n";
        }
}
```

When we pass references into a subroutine we're allowing that subroutine full access to the structure that the reference refers to. All changes that the subroutine makes to that structure will remain after the subroutine has returned. If you wish to make a copy of the structure that the reference refers to and modify that locally, you can do the following:

```
sub ref_print_unchosen {
        my ($chosen, $colours) = @_;

        my @chosen = @$chosen;          # this @chosen is now a copy
        my @colours = @$colours;        # this @colours is now a copy
}
```

The above paragraph discusses a concept that is often referred to as *call by reference*. Typically when we call Perl subroutines we consider them to be called *by value*. Technically, however, this is incorrect.

In the case where we pass scalars into a subroutine, we usually `shift` them from `@_` or we copy the contents from `@_` into another list. However if we instead modify the contents of `@_` directly we will actually be modifying the contents of the variables given to the subroutine.

We don't recommend this practice, however, as it makes your code much harder for other people to maintain. It's much better to do something like the following:

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

If you do use this functionality, be careful, as it's a fatal error to attempt to modify a read-only value, such as a literal string.

Anonymous data structures

We can use anonymous data structures to create complex data structures without having to declare many temporary variables. Anonymous arrays are created by using square brackets instead of round ones. Anonymous hashes use curly braces instead of round ones.

```
# the old two-step way:
my @array = qw(a b c d);
my $array_ref = \@array;

# if we get rid of $array_ref, @array will still hang round using up
# memory.  Here's how we do it without the intermediate step, by
# creating an anonymous array:

my $array_ref = ['a', 'b', 'c', 'd'];

# look, we can still use qw() too...

my $array_ref = [qw(a b c d)];

# more useful yet, we can put these anon arrays straight into a hash:

my %transport = (
        'cars'          =>   [qw(toyota ford holden porsche)],
        'planes'        =>   [qw(boeing harrier)],
        'boats'         =>   [qw(clipper skiff dinghy)],
);
```

The same technique can be used to create anonymous hashes:

```
# The old, two-step way:
my %hash = (
        a       =>   1,
        b       =>   2,
        c       =>   3
);
my $hash_ref = \%hash;

# the quicker way, with an anonymous hash:
my $hash_ref = {
        a       =>   1,
        b       =>   2,
        c       =>   3
};
```

Data is pulled out of an anonymous data structure using the arrow notation:

```
my $value = $hash_ref->{a};
print $value;              # prints 1
```

Exercise

1. Change your previous program to initialise `$friends_ref` using an anonymous array constructor. You should no longer need your original `@friends` array. Test that your program still works.

Complex data structures

References are most often used to create complex data structures. Since references are scalars, they can be used as values in both hashes and arrays. This makes it possible to create both deep and complex multi-dimensional data structures. These are covered more deeply in Appendix C.

The use of references in data structures allows you to create arrays of arrays, arrays of hashes, hashes of arrays and hashes of hashes. We saw an example of a hash of arrays in the previous section. Here is an example of an array of hashes:

```
my %alice = (
                name => "Alice Jane",
                age => 34,
                employeenumber => 12003,
);
my %bob = (
                name => "Bob Jane",
                age => 32,
                employeenumber => 12345,
);

my @employees = (
        \%alice,
        \%bob,
);

# to print out Alice's employee number:
print $employees[0]->{employeenumber};
```

```
# Or, to use anonymous data structures
my @employees2 = (
        {
                name => "Alice Jane",
                age => 34,
                employeenumber => 12003,
        },
        {
                name => "Bob Jane",
                age => 32,
                employeenumber => 12345,
        },
);

# to print out Bob's age:
print $employees2[1]->{age};
```

Exercises

1. Create data structures as follows:
    a. Create a hash called `%pizza_prices` which contains prices for small, medium and large pizzas.
    b. Create a hash called `%pasta_prices` which contains prices for small, medium and large serves of pasta.
    c. Create a hash called `%milkshake_prices` which contains prices for small, medium and large milkshakes.
    d. Create a hash called `%menu` containing references to the above hashes, so that given a type of food and a size you can find the price of it. Don't forget that your hash must contain both keys (the type of food), and values (a reference to the data structure containing the prices).
2. Write code which accepts food-type and size from the user and returns the price for the food.
3. Convert the above hash to use anonymous data structures (to get a hash of hashes) instead of the original three pizza, pasta and milkshake hashes, and modify your customer code appropriately.
4. Add a new element to your foods hash which contains the prices of salads. Rather than adding this in when you create the hash, instead add it separately.
5. Create a subroutine which can be passed a scalar and a hash reference. Check whether there is an element in the hash which has the scalar as its key. Hint: use `exists` for this.

Answers for the above exercises can be found in `exercises/answers/food.pl`.

Disambiguation and curly braces

Often in our code, we need to treat a reference as its underlying data structure. For a simple reference, this is easy; we prepend the reference with the appropriate sigil and it just works:

```
my $hashref = { a => 1, b => 2, c => 4, d => 8 };

foreach ( keys %$hashref ) {
        ...
}
```

What can cause us problems is when the reference isn't so simple. What should Perl do, in the following case?

```
my @result = @$array[0];
```

Does this mean:

- Find `@array`.
- Look up index `0`: `$array[0]`
- Turn that (`$array[0]`) into an array: `@$array[0]`

or:

- Find the array reference `$array`
- Treat that as an array: `@$array`
- Take an array slice with index `0`: `@$array[0]`

Perl does the latter, however if that is what we wanted then we should have written `$$array[0]`, as that explicitly returns a single (scalar) result.

We can force Perl to evaluate our expression as the first interpretation above by using curly braces. This allows us to clearly write:

```
my @result = @{$array[0]};
```

We can use `${...}`, `@{...}` or `%{...}` syntax to evaluate any expression and dereference the result.

Data::Dumper

Typically, to print out a data structure you have to understand its underlying structure and then write a number of loops to print it out in full. If the structure is relatively simple such as a hash of hashes of values, or even a hash of hash of arrays this isn't too difficult.

However, often data structures are very complex, and negotiating and printing these structures can be a tiresome exercise. It's also an unnecessary one, as all the hard work has already been done for you. To save you from having to write specialised printing code in every program for debugging purposes, there's a special library you may find useful called `Data::Dumper`.

`Data::Dumper` provides a function which takes Perl data structures and turns them into human readable strings representing the data with in them. It can be used just like this:

```
#!/usr/bin/perl -w
use strict;
use Data::Dumper;
```

```
my %HoH = (
        Jacinta => {
                        age => 26,
                        favourite_colour => "blue",
                        sport => "swimming",
                        language => "Perl",
                    },


        Paul => {
                        age => 27,
                        favourite_colour => "green",
                        sport => "cycling",
                        language => "Perl",
                },
);

print Dumper \%HoH;
```

This will print out something similar to:

```
$VAR1 = {
          'Paul' => {
                      'language' => 'Perl',
                      'favourite_colour' => 'green',
                      'sport' => 'cycling',
                      'age' => 27
                    },
          'Jacinta' => {
                         'language' => 'Perl',
                         'favourite_colour' => 'blue',
                         'sport' => 'swimming',
                         'age' => 26
                       }
        };
```

Not only is this easy to read, but it's also perfectly valid Perl code. This means you can use `Data::Dumper` to easily give you a structure that you can paste into another program, or which can be 'serialised' to a file and re-created at a later date. `Data::Dumper` has a lot more uses beyond simple debugging.

`Dumper` expects to be given one or more references to data structures to dump. If `Dumper` is provided with a hash or array then every element of the array, or every key and value of the hash, will be considered a separate data structure, and dump separately. The results are not particularly useful:

```
# result of: print Dumper %HoH;
$VAR1 = 'Paul';
$VAR2 = {
          'language' => 'Perl',
          'favourite_colour' => 'green',
          'sport' => 'cycling',
          'age' => 27
        };
$VAR3 = 'Jacinta';
$VAR4 = {
          'language' => 'Perl',
          'favourite_colour' => 'blue',
          'sport' => 'swimming',
          'age' => 26
        };
```


You can read more about `Data::Dumper` on page 882 of the Camel book or in **perldoc Data::Dumper**.

Exercises

1. Use `Data::Dumper` to print out your data structures from the previous exercise.
2. Use **perldoc Data::Dumper** to read about `Data::Dumper`'s many options and configuration variables.

Chapter summary

- References are scalar data consisting of the memory address of a piece of Perl data, and can be used in arrays, hashes, and other places where you would use a normal scalar
- References can be used to create complex data structures, to pass multiple arrays or hashes to subroutines, and in object-oriented Perl.
- References are created by prepending a backslash to a variable name.
- References are dereferenced by replacing the name part of a variable name (eg `foo` in `$foo`) with a reference, for example replace `foo` with `$foo_ref` to get `$$foo_ref`
- References to arrays and hashes can also be dereferenced using the arrow `->` notation.
- References can be passed to subroutines as if they were scalars.
- References can be included in arrays or hashes as if they were scalars.
- Anonymous arrays can be made by using square brackets instead of round; anonymous hashes can be made by using curly brackets instead of round. These can be assigned directly to a reference, without any intermediate step.
- `Data::Dumper` allows complex data structures to be printed out verbatim without requiring full knowledge of the underlying data structure.

Chapter 12. External Files and Packages

In this chapter...

In this chapter we'll discuss how we can split our code into separate files. We'll discover Perl's concept of packages, and how we can use them to make our code more robust and flexible.

Splitting code between files

When writing small, independent programs, the code can usually be contained within a single file. However there are two common occurrences where we would like to have our programs span multiple files. When working on a large project, often with many developers, it can be very convenient to split a program into smaller files, each with a more specialised purpose. Alternatively, we may find ourselves working on many programs that share some common code base. This code can be placed into a separate file which can be shared across programs. This saves us time and effort, and means that bug-fixes and improvements need to be made only in a single location.

Require

Perl implements a number of mechanisms for loading code from external files. The most simplest of these is by using the `require` function:

```
require 'file.pl';
```

Perl is smart enough to make sure that the same file will not be included twice if it's required through the same specified name.

```
# The file is only included once in the following case:
require 'file.pl';
require 'file.pl';
```

Required files *must* end with a true value. This is usually achieved by having the final statement of the file being:

```
1;
```

Conflicts can occur if our included file declares subroutines with the same name as those that appear in our main program. In most circumstances the subroutine from the included file takes precedence, and a warning is given.

We will learn how to avoid these conflicts later in this chapter when we discuss the concept of *packages*.

The use of `require` has been largely deprecated by the introduction of modules and the `use` keyword. If you're writing a code library from scratch we recommend that you create it as a module. However, `require` is often found in legacy code and is a useful thing to understand.

Any code in the file (except for subroutines) will be executed immediately when the file is required. The `require` occurs at run-time, this means that Perl will not throw an error due to a missing file until that statement is reached, and any subroutines inside the file will not be accessible until after the `require`.

Variables declared with `my` are not shared between files, they are only visible inside the block or file where the declaration occurs. To share packages between files we use *package variables* which are covered later in this chapter.

The use of modules (which we will learn about later) allows for external files to be loaded at compile-time, rather than run-time.

Use strict and warnings

Perl pragmas, such as `strict` and `warnings` are lexically scoped. Just like variables declared with `my`, they last until the end of the enclosing block, file or eval.

This means that you can turn strict and warnings on in one file without it influencing other parts of your program. Thus, if you're dealing with legacy code, then your new libraries, modules and classes can be strict and warnings compliant even though the older code is not.

Example

The use of `require` is best shown by example. In the following we specify two files, `Greetings.pl` and `program.pl`. Both are valid Perl programs on their own, although in this case, `Greetings.pl` would just declare a variable and a subroutine, and then exit. As we do not intend to execute `Greetings.pl` on its own, it does not need to be made executable, or include a shebang line.

Our library code, to be included.

```
# Greetings.pl
# Provides the hello() subroutine, allowing for greetings
# in a variety of languages.  English is used as a default
# if no language is provided.

use strict;
use warnings;

my %greeting = (
        en      => "Hello",
        'en-au' => "G'day",
        fr      => "Bonjour",
        jp      => "Konichiwa",
        zh      => "Nihao",
);
```

```
sub hello {
        my $language = shift || "en";

        my $greeting = $greeting{$language}
                or die "Don't know how to greet in $language";

        return $greeting;
}

1;
```

Our program code.

```
# program.pl
# Uses the Greetings.pl file to provide another hello() subroutine
use strict;

# Get the contents from file.pl
require "Greetings.pl";

print "English: ",    hello("en"),   "\n";       # Prints "Hello"
print "Australian: ", hello("en-au"),"\n";       # Prints "G'day"
```

Exercises

1. Create a file called `MyTest.pl` Define at least two functions; `pass` and `fail` which print some amusing output. Make sure that it uses `strict`.
2. Test that your code compiles by running **perl -c MyTest.pl**. (The **-c** tells Perl to check your code).
3. Create a simple Perl script which requires `MyTest.pl` and calls the functions defined within.

Introduction to packages

The primary reason for breaking code into separate files is to improve maintainability. Smaller files are easier to work with, can be shared between multiple programs, and are suitable for dividing between members of large teams. However they also have their problems.

When working with a large project, the chances of naming conflicts increases. Two entirely different files may have two different subroutines with the same name; however it is only the last one loaded that will be used by Perl. Files from different projects may be re-used in new developments, and these may have considerable name clashes. Multiple files can also make it difficult to determine where subroutines are originally declared, which can make debugging difficult.

Perl's *packages* are designed to overcome these problems. Rather than just putting code into separate files, code can be placed into independent packages, each with its own namespace. By ensuring that package names remain unique, we also ensure that all subroutines and variables can remain unique and easily identifiable.

A single file can contain multiple packages, but convention dictates that each file contains a package of the same name. This makes it easy to quickly locate the code in any given package.

Writing a package in Perl is easy. We simply use the `package` keyword to change our current package. Any code executed from that point until the end of the current file or block is done so in the context of the new package.

```
# By declaring that all our code is in the "Greetings" package,
# we can be certain not to step on anyone else's toes, even if
# they have written a hello() subroutine.

package Greetings;

use strict;
use warnings;

my %greeting = (
        en      => "Hello",
        'en-au' => "G'day",
        fr      => "Bonjour",
        jp      => "Konichiwa",
        zh      => "Nihao",
);

sub hello {
        my $language = shift || "en";

        my $greeting = $greeting{$language}
                or die "Don't know how to greet in $language";

        return $greeting;
}

1;
```

The package that you're in when the Perl interpreter starts (before you specify any package) is called `main`. Package declarations use the same rules as `my`, that is, it lasts until the end of the enclosing block, file, or eval.

Perl convention states that package names (or each part of a package name, if it contains many parts) starts with a capital letter. Packages starting with lower-case are reserved for pragmas (such as `strict`).

The scoping operator

Being able to use packages to improve the maintainability of our code is important, but there's one important thing we have not yet covered. How do we use subroutines, variables, or filehandles from other packages?

Perl provides a *scoping operator* in the form of a pair of adjacent colons. The scoping operator allows us to refer to information inside other packages, and is usually pronounced "double-colon".

```
require "Greetings.pl"

# This calls the hello() subroutine in our main package,
# printing "Greetings Earthling".
print hello(),"\n";

# Greetings in English.
print Greetings::hello("en"),"\n";

# Greetings in Japanese.
print Greetings::hello("jp"),"\n";
```

```
sub hello {
        return "Greetings Earthling";
}
```

Calling subroutines like this is a perfectly acceptable alternative to exporting them into your own namespace (which we'll cover later). This makes it very clear where the called subroutine is located, and avoids any possibility of an existing subroutine clashing with that from another package.

Occasionally we may wish to change the value of a variable in another package. It should be very rare that we should need to do this, and it's not recommended you do so unless this is a documented feature of your package. However, in the case where we do need to do this, we use the scoping operator again.

```
use Carp;

# Turning on $Carp::Verbose makes carp() and croak() provide
# stack traces, making them identical to cluck() and confess().
# This is documented in 'perldoc Carp'.

$Carp::Verbose = 1;
```

There's a shorthand for accessing variables and subroutines in the `main` package, which is to use double-colon without a package name. This means that `$::foo` is the same as `$main::foo`.

When referring to a variable in another package, the sigil (punctuation denoting the variable type) always goes *before* the package name. Hence to get to the scalar `$bar` in the package `Foo`, we would write `$Foo::bar` and not `Foo::$bar`.

It is not possible to access lexically scoped variables (those created with `my`) in this way. Lexically scoped variables can *only* be accessed from their enclosing block.

Package variables and our

It is not possible to access lexically scoped variables (those created with `my`) outside of their enclosing block. This means that we need another way to create variables to make them globally accessible. These global variables are called *package variables*, and as their name suggests they live inside their current package. The preferred way to create package variables, under Perl 5.6.0 and above, is to declare them with the `our` statement. Of course, there are alternatives you can use with older version of Perl, which we also show here:

```
package Carp;

our $VERSION = '1.01';          # Preferred for Perl 5.6.0 and above

# use vars qw/$VERSION/;        # Preferred for older versions
# $VERSION = '1.01';

# $Carp::VERSION = '1.01';      # Acceptable but requires that we then
                                # always use this full name under strict
```

In all of the cases above, both our package and external code can access the variable using `$Carp::VERSION`.

Exercises

1. Change your `MyTest.pl` file to include a package name `MyTest`
2. Update your program to call the MyTest functions using the scoping operator.
3. Create a package variable `$pass_mark` using `our` inside `MyTest.pl` which defines an appropriate pass mark.
4. In your Perl script, create a loop which tests 10 random numbers for pass or fail with reference to the `$pass_mark` package variable. Print the appropriate `pass` or `fail` message.
5. Print out the version of the `Cwd` module installed on your training server. The version number is in `$Cwd::VERSION`.
6. Look at the documentation for the `Carp` module using the **perldoc Carp** command. This is one of Perl's most frequently used modules.

Answers for the above exercises can be found in `exercises/answers/MyTest.pl` and `exercises/answers/packages.pl`.

Chapter summary

- A package is a separate namespace within Perl code.
- A file can have more than one package defined within it.
- The default package is `main`.
- We can get to subroutines and variables within packages by using the double-colon as a scoping operator for example `Foo::bar()` calls the `bar()` subroutine from the `Foo`
- To write a package, just write `package` *`package_name`* where you want the package to start.
- Package declarations last until the end of the enclosing block, file or eval (or until the next package statement).
- Package variables can be declared with the `our` keyword. This allows them to be accessed from inside other packages.
- The `require` keyword can be used to import the contents of other files for use in a program.
- Files which are included using `require` must end with a true value.

Chapter 13. Modules

In this chapter...

In this chapter we'll discuss modules from a user's standpoint. We'll find out what a module is, how they are named, and how to use them in our work.

In the remainder of the chapter, we will investigate how to write our own modules.

Module uses

Perl modules can do just about anything. In general, however, there are three main uses for modules:

- Changing how the rest of your program is interpreted. For example, to enforce good coding practices (`use strict`) or to allow you to write in other languages, such as Latin (`use Lingua::Romana::Perligata`), or to provide new language features (`use Switch`).
- To provide extra functions to do your work (`use Carp` or `use CGI qw/:standard/`).
- To make available new classes (`use HTML::Template` or `use Finance::Quote`) for object oriented programming.

Sometimes the boundaries are a little blurred. For example, the `CGI` module provides both a class and the option of extra subroutines, depending upon how you load it.

What is a module?

A module is a separate file containing Perl source code, which is loaded and executed at compile time. This means that when you write:

```
use CGI;
```

Perl looks for a file called `CGI.pm` (.pm for *Perl Module*), and upon finding it, loads it in and executes the code inside it, before looking at the rest of your program.

Sometimes you need to tell Perl where to look for your Perl modules, especially if some of them are installed in a non-standard place. Like many things in Perl, There's More Than One Way To Do It. Check out **perldoc -q library** for some of the ways to tell Perl where your modules are installed.

Sometimes you might choose to pass extra information to the module when you load it. Often this is to request the module create new subroutines in your namespace.

```
use CGI qw(:standard);
use File::Copy qw(copy);
```

Note the use of `qw()`, this is a list of words (in our case, just a single word). It's possible to pass many options to a module when you load it. In the case above, we're asking the `CGI` module for the `:standard` bundle of functions, and the `File::Copy` module for just the `copy` subroutine.

Each module has a different set of options (if any) that it will accept. You need to check the documentation of the module you're dealing with to which (if any) are applicable to your needs.

To find out what options exist on any given module read its documentation: **perldoc** ***module_name***.

Exercise

1. Using `File::Copy` make a copy of one of your files. If you're eager, ask the user which file to copy and what to name the copy.

Where does Perl look for modules?

Perl searches through a list of directories that are determined when the Perl interpretor is compiled. You can see this list (and all the other options Perl was compiled with), by using **perl -V**.

The list of directories which Perl searches for modules is stored in the special variable `@INC`. It's possible to change `@INC` so that Perl will search in other directories as well. This is important if you have installed your own private copy of some modules.

Of course, being Perl, there's more than one way to change `@INC`. Here are some of the ways to add to the list of directories inside `@INC`:

- Call Perl with the `-I` command-line switch with the location of the extra directory to search. For example:

  ```
  perl -I/path/to/libs
  ```

  This can be done either in the shebang line, or on the command-line.
- Use the `lib` pragma in your script to inform Perl of extra directories. For example:

  ```
  use lib "/path/to/libs";
  ```
- Setting the `PERL5LIB` environment variable with a colon-separated list of directories to search. Note that if your script is running with taint checks this environment variable is ignored.

Since `use` statements occur before regular Perl code is executed, modifying `@INC` directly usually does not have the desired effect.

Finding installed modules

Perl comes with many modules in its standard distribution. You can get a list of all of them by doing a **perldoc perlmodlib**. The Camel book describes the standard modules in chapters 31 and 32 (chapter 7, 2nd Ed).

Besides from the modules in the standard distribution, you can also see any other modules that were installed on your system by using **perldoc perllocal**. Generally this file only lists other modules that were installed by hand, or using one of the CPAN installers (more on this later).

> Modules installed through your operating system's packaging system may not appear in **perldoc perllocal**.

You can get more information on any module that you have installed by using **perldoc** ***module_name***. For example, **perldoc English** will give you information about the `English` module.

Most importantly, there's a great resource for finding modules called the *Comprehensive Perl Archive Network*, or *CPAN* for short. The CPAN website (http://www.cpan.org/) provides many ways of finding the modules you're after and browsing their documentation on-line. It's highly recommended that you become familiar with CPAN's search features, as many common problems have been solved and placed in CPAN modules.

Exercise

1. Open a web browser to CPAN's search site (http://search.cpan.org) and spend a few minutes browsing the categories provided.
2. Perform a search on CPAN for a problem domain of your choice. If you can't think of one, search on `CGI`, `XML` or `SOAP`.

Using CPAN modules

CPAN provides more than 9,000 separate and freely available modules. This makes CPAN an excellent starting point when you wish to find modules to help solve your particular problem. However, you should keep in mind that not all CPAN modules are created equal. Some are much better documented and written than others. Some (such as the `CGI` or `DBI`) modules have become de-facto standards, whereas others may not be used by anyone except the module's author.

As with any situation when you're using third party code, you should take the time to determine the suitability of any given module for the task at hand. However, in almost all circumstances it's better to use or extend a suitable module from CPAN rather than trying to re-invent the wheel.

Many of the popular CPAN modules are pre-packaged for popular operating systems. In addition, the `CPAN` module that comes with Perl can make the task of finding and installing modules from CPAN much easier.

Most CPAN modules come with `README` and/or `INSTALL` files which tell you how to install the modules. This may vary between operating systems. On Unix and Unix-like operating systems the process is usually:

```
perl Makefile.PL
make
make test
make install
```

For ActiveState Perl installations (which includes most Microsoft Windows machines) the use of PPM (Programmer's Package Manager) is recommended. PPM provides a command line interface for downloading and installing pre-compiled versions of most CPAN modules.

Some times you may not find the module you're looking for through PPM. In this case you may want to build your own. The process for this is similar to that for Unix machines, although instead of using **make** you will need to use **nmake** which is a `make` equivalent made by Microsoft. Some Perl modules also require a C compiler.

Some times you may not be able to, or may not wish to, install CPAN modules in their default path. In this case you can provide a flag to the `Makefile.PL` program instructing it on your preferred top level directory. For example:

```
perl Makefile.PL PREFIX=/home/sue/perl/
```

If you install your module in a different directory than your other Perl modules you may have to use the `lib` pragma, mentioned in the previous section, to tell Perl where to find your files. Once a module is installed, you can use it just like any other Perl module.

The double-colon

Sometimes you'll see modules with double-colons in their names, like `Finance::Quote`, `Quantum::Superposition`, or `CGI::Fast`. The double-colon is a way of grouping similar modules together, in much the way that we use directories to group together similar files. You can think of everything before the double-colon as the category that the module fits into.

In fact, the file analogy is so true-to-life that when Perl searches for a module, it converts all double-colons to your directory separator and then looks for that when trying to find the appropriate file to load. So `Finance::Quote` looks for a file named `Quote.pm` in a directory called `Finance`. That two modules are in the same category doesn't necessarily mean that they're related in any way. For example, `Finance::Quote` and `Finance::QuoteHist` have very similar names, and their maintainers even enjoy very similar hobbies, they certainly have similar uses, but neither package shares any code in common with the other.

It's perfectly legal to have many double-colon separators in module names, so `Chicken::Bantam::SoftFeather::Pekin` is a perfectly valid module name.

Writing modules

Modules contain regular Perl code, and for most modules the vast majority of that code is in subroutines. Sometimes there are a few statements which initialise variables and other things before any of those subroutines are called, and those get executed immediately. The subroutines get compiled and tucked away for later use.

Besides from the code that's loaded and executed, two more special things happen. Firstly, if the last statement in the module did not evaluate to true, the Perl compiler throws an exception (usually halting your program before it even starts). This is so that a module could indicate that something went wrong, although in reality this feature is almost never used. Virtually any Perl module you care to look at will end with the statement `1;` to indicate successful loading.

The other thing that happens when a module is `use`d is that its `import` subroutine (if one exists) gets called with any directives that were specified on the `use` line. This is useful if you want to export functions or variables to the program that's using your module for functional programming but is almost never used (and very often discouraged) for object oriented programming.

As you've no doubt guessed by now, modules and packages often go hand-in-hand. We know how to use a module, but what are the rules on writing one? Well, the big one is this:

> A module is a file that contains a package of the same name.

That's it. So if you have a package called `Tree::Fruit::Citrus::Lime`, the file would be called `Tree/Fruit/Citrus/Lime.pm`, and you would use it with `use Tree::Fruit::Citrus::Lime;`.

A module can contain multiple packages if you desire. So even though the module is called `Chess::Piece`, it might also contain packages for `Chess::Piece::Knight` and `Chess::Piece::Bishop`. It's usually preferable for each package to have its own module, otherwise it can be confusing to your users how they can load a particular package.

When writing modules, it's important to make sure that they are well-named, and even more importantly that they won't clash with any current or future modules, particularly those available via CPAN. If you are writing a module for internal use only, you can start its name with `Local::` which is reserved for the purpose of avoiding module name clashes.


You can read more about writing modules in **perldoc perlmodlib**, and a little on pages 554-556 of the Camel book.

Use versus require

Perl offers several different ways to include code from one file into another. `use` is built on top of `require` and has the following differences:

- Files which are `used` are loaded and executed at compile-time, not run-time. This means that all subroutines, variables, and other structures will exist before your main code executes. It also means that you will immediately know about any files that Perl could not load.
- `use` allows for the import of variables and subroutines from the used package into the current one. This can make programming easier and more concise.
- Files called with `use` can take arguments. These arguments can be used to request special features that may be provided by some modules.

Both methods:

- Check for redundant loading, and will skip already loaded files.
- Raise an exception on failure to find, compile or execute the file.
- Translate `::` into your systems directory separator.

Where possible `use` and Perl *modules* are preferred over `require`.

Warnings and strict

When your module is used by a script, whether or not it runs with warnings depends upon whether the calling script is running with warnings turned on. You can (and should) invoke the `use warnings` pragma to turn on warnings for your module without changing warnings for the calling script.

Your modules should always use strict.

```
use strict;
use warnings;
```

Exercise

This exercise will have you adapt your `MyTest.pl` code to become a module. There's a list at the end of this exercise of things to watch out for.

1. Create a directory named `lib`.
2. Move your `MyTest.pl` file into your `lib` directory and rename it to `MyTest.pm`.
3. Make sure `MyTest.pm` uses `strict` and `warnings`.
4. Test that your module has no syntax errors by running **perl -c MyTest.pm**.
5. Change your Perl script from before to use the `lib` pragma in order to find your module. (`use lib 'lib';`)
6. Change your Perl script to `use` your module. Check that everything still works as you expect.
7. Add a print statement to your module (outside any subroutines). This should be printed when the module is loaded. Check that this is so.

Answers can be found in `exercises/answers/lib/MyTest.pm` and `exercises/answers/modules.pl`

Things to remember...

The above exercises can be completed without reference to the following list. However, if you're having problems, you may find your answer herein.

- A module is a file that contains a package of the same name.
- Perl modules must return a true value to indicate successful loading. (Put `1;` at the end of your module).
- To use a module stored in a different directory, add this directory to the `@INC` array. (Put `use lib 'path/to/modules/'` before the other `use` lines.
- To call a subroutine which is inside a module, you can access it via the double-colon. Eg: `MyModule::test();`

Exporting and importing subroutines

Writing your own `import` function for each and every module would be a tiresome and error-prone process. However, Perl comes with a module called `Exporter`, which provides a highly flexible interface with optimisations for the common case.

`Exporter` works by checking inside your module for three special data structures, which describe both what you wish to export, and how you wish to export them. These structures are:

- `@EXPORT` symbols to be exported into the user's name space by default
- `@EXPORT_OK` symbols which the user can request to be exported
- `%EXPORT_TAGS` that allows for bundles of symbols to be exported when the user requests a special export tag.

@ISA

To take advantage of `Exporter`'s `import` function we need to let Perl know that our package has a special relationship with the `Exporter` package. We do this by telling Perl that we *inherit* from `Exporter`. Our package and the rest of our program does not need to be written in an object oriented style for this to work.

Now when Perl goes looking for the `import` function it will first look in our package. If it can't be found there, Perl will look for a special array called `@ISA`. The contents of the `@ISA` array is interpreted as a list of parent classes, and each of these will be searched for the missing method.

To specify that this package is a sub-class of the Exporter module we include the following lines:

```
use Exporter;
our @ISA = qw(Exporter);
```

use base

An alternative to adding parent modules to `@ISA` yourself is to use the `base` pragma. This allows you to declare a derived class based upon the listed parent classes. Thus the two lines above becomes:

```
use base qw(Exporter);
```

The `base` pragma takes care of ensuring that the `Exporter` module is loaded.

The `base` pragma is available for all versions of Perl above 5.6.0.

An example

Here's an example of just using `@EXPORT` and `@EXPORT_OK`. Our hypothetical module, `People::Manage` is used for managing interpersonal relations.

```
package People::Manage;
use base qw(Exporter);
use vars qw(@EXPORT @EXPORT_OK);

@EXPORT     = qw(invite $name @friends %addresses);        # invite is a subroutine
@EXPORT_OK  = qw(&taunt $spouse @enemies %postcodes);      # so is taunt
```

The ampersand in front of subroutines is optional.

Exporting by default

Exporting your symbols by default, by populating the `@EXPORT` array, means that anyone using your module will receive these symbols without having to ask for them. This is generally considered to be bad style, and is sometimes referred to as 'polluting' the caller's namespace.

The reason this is considered to be bad style is that there is nothing in the `use` line to indicate that anything is being exported. A programmer who is not familiar with the module may inadvertently define their own subroutines or variables which clash with those that are exported. Likewise, a reviewer examining the code will not easily be able to determine from which module a given subroutine may have been exported, especially if many modules are used.

Using the `@EXPORT` array is highly discouraged.

Using `@EXPORT_OK` allows the user to choose which symbols they wish to bring into their name space. All other symbols can be accessed by using their full name, such as `People::Manage::invite()`, when required.

An example

Our module:

```
###### People/Manage.pm ############
package People::Manage;         # create a package of the same name
use strict;
use warnings;
use base qw(Exporter);

# List out the things we wish to export
our @EXPORT_OK  = qw(invite $name @friends %addressbook
                     taunt $spouse @enemies @children $pet);

# Only package variables can be exported, as such all of these
# variables need to be declared with 'our' not 'my'.

our $name   = "Fred";
our $spouse = "Wilma";

our @children = qw(Pebbles);
our @friends  = qw(Barney Betty);
our $pet      = "Dino";
my $address = "301 CobbleStone Way, Bedrock";

our %addressbook = (
        Barney         => "303 Cobblestone Way, Bedrock",
        Betty          => "303 Cobblestone Way, Bedrock",
        "Barney's Mom" => "142 Boulder Ave, Granitetown",
);


sub invite {
        my ($friend, $date) = @_;
        return "Dear $friend,\n $spouse and I would love you to come to".
               "dinner at our place ($address) on $date.\n\n".
               "Yours sincerely, $name\n";
}

sub taunt {
        my ($enemy) = @_;
        return "Dear $enemy, my pet $pet has more brains than you.\n";
}

1;                              # module MUST end with something true
```

Our program:

```
###### dinner.pl ############
#!/usr/bin/perl -w
use strict;
use People::Manage qw(invite %addressbook);     # only using a few things

# Invite some people over for dinner.
```

```
foreach my $person (keys %addressbook) {
        print invite($person,"next Tuesday");
}
```

Importing symbols

Once your module is written and it exports a few symbols, it's time to use it. This is done with the `use` command that we've seen with `strict` and other modules. We can load our module in three ways:

- `use People::Manage;` which imports all of the symbols stored in `@People::Manage::EXPORT`.
- `use People::Manage ();` which imports *none* of the symbols in either `@People::Manage::EXPORT` or `@People::Manage::EXPORT_OK`.
- `use People::Manage qw($name $spouse invite);` which imports all the listed symbols. If a symbol is mentioned which is not in either `@People::Manage::EXPORT` or `@People::Manage::EXPORT_OK` then a fatal error will occur.

Exercises

These exercises build on the previous exercises.

1. Change your MyTest.pm module to export the `pass` and `fail` symbols and import those into your script. Change your script to call `pass` and `fail` instead of their fully qualified names.
2. Change your module to export the `$pass_mark` variable and use that instead of its fully qualified name.

Exporting tags

If you wish to export groups of symbols that are related to each other, there is an `%EXPORT_TAGS` hash which provides this functionality. This can be used in the follow manner:

```
%EXPORT_TAGS = (family => [ qw/$name $spouse @children $pet/ ],
                social => [ qw/%address invite taunt @friends/ ],
);
```

Names which appear in `%EXPORT_TAGS` must also appear in `@EXPORT` or `@EXPORT_OK`. Tags themselves cannot be used in either export array.

Importing symbols through tags

Symbols grouped in tags can be imported normally, by specifying each symbol, or by using the tag provided. This is done by prepending the tag name with a colon:

```
use People::Manage qw/:family/;           # Family related information.
# use People::Manage qw/:social/;         # Social-related symbols.
# use People::Manage qw/:family :social/; # Both
```

Exercise

1. In your `MyTest` module, create a tag which contains both subroutines and use that instead of specifying them both during the import.

Chapter summary

- A module is a separate file containing Perl source code.
- We can use modules by writing `use module_name;` before we want to start using it.
- Perl looks for modules in a list of directories that are determined when the Perl interpretor is compiled.
- Module names may contain double-colons (`::`) in their names such as `Finance::Quote`, these tell where Perl to look for a module (in this case in the `Finance/` directory.
- Modules can be used for class definitions or as libraries for common code.
- A module can contain multiple packages, but this is often a bad idea.
- It's often a good idea to put your own modules into the `Local` namespace.

Chapter 14. Using Perl objects

In this chapter...

While discussion of Object Oriented programming is beyond the scope of this course, a great many modules you may encounter while programming provide an object oriented interface. This chapter will teach you what you need to know to use these modules.

 Perl Training Australia runs a two day course on Object Oriented Programming in Perl, for more information visit our website (http://www.perltraining.com.au/) or talk to your instructor during a break.

Objects in brief

An *object* is a collection of data (attributes) and subroutines (methods) that have been bundled into a single item. Objects often represent real-world concepts, or logical constructs. For example, an *invoice* object may have attributes representing the date posted, date date, amount payable, GST, vendor, and so on. The invoice may have various methods that allow for payment to be made, and possibly a payment to allow the invoice to be disputed.

An object in Perl is a reference to a specially prepared data structure. This structure may be a hash, an array, a scalar or something more complex. However, as the user of an object, we don't need to know (and should not care) what sort of structure is actually being used. What matters are the *methods* on the object, and how we can use them.

A Perl object is a *special* kind of reference because it also knows what class it belongs to. In other words, an object knows what kind of object it is.

Object orientation allows us to create *multiple* objects from the same class which can each store different information and behave differently according to that information. This makes it very easy for the users of those objects, as it makes the information easy to track and manipulate.

Using an object

To use a Perl module which provides an object oriented interface we `use` it without specifically importing any methods. For our examples we will use the `DBI` module, which allows us to interact with a number of databases, and is one of the most commonly used modules in Perl.

```
#!/usr/bin/perl -w
use strict;
use DBI;               # We can now create DBI objects.
```

 To learn more about `DBI` read **perldoc DBI** and the `DBI` homepage (http://dbi.perl.org/).

Perl Training Australia also runs a Database Programming with Perl course which you may find of interest. For more information visit our website (http://www.perltraining.com.au/) or talk to your instructor during the break.

Instantiating an object

To create a new object we call the constructor method on the *name* of the class. In many cases this method is called `new`, however with `DBI` it is called `connect`; as we get our database handle by *connecting* to a database.

```
use DBI;

# Create a DBI object (database connection handle)
my $dbh = DBI->connect($data_source, $username, $password);
```

By convention, our connected database object is called `$dbh`, for "database handle".

We can create a number of database handles (objects), with each connecting to different databases or with different usernames and passwords. We could also create a number of database handles connecting to the same database. This could potentially be useful if we wished to execute multiple SQL commands simulatenously, particularly if we're dealing with a clustered database system.

```
use DBI;

my $oracle_dbh   = DBI->connect($oracle_dsn, $oracleuser, $oraclepasswd);
my $postgres_dbh = DBI->connect($postgres_dsn, $postgresuser, $postgrespasswd);
my $mysql_dbh1   = DBI->connect($mysql_dsn, $mysqluser1, $mysqlpasswd1);
my $mysql_dbh2   = DBI->connect($mysql_dsn, $mysqluser2, $mysqlpasswd2);
```

Each of these objects represent a different database connection and we can call the other `DBI` methods on these objects from now on. Each object will remember which database it refers to without further work on behalf of the programmer.

Calling methods on an object

As we covered earlier, we can get at the contents of a normal reference by using the arrow operator:

```
$array_ref->[$index];          # Access array element via array reference
$hash_ref->{$key};             # Access array element via hash reference
```

It should come as no big surprise that Perl object methods (or functions, if you'd prefer) can be accessed the same way:

```
$object->method();             # Call method() on $object
```

In a specific case, we can call a method on one of our `DBI` objects as follows:

```
use DBI;

my $dbh = DBI->connect($data_source, $username, $password);

$dbh->do("UPDATE friends SET phone = '12345678' WHERE name = 'Jack'");
```

Destroying an object

When you no longer need an object you can let it go out of scope, just as when you no longer need any other Perl data structure. In some cases the documentation may recommend calling certain clean up functions. In the case of `DBI` it is considered polite to disconnect from the database.

```
$dbh->disconnect();
```

Chapter summary

- Perl objects are special references to Perl data structures which know which class they belong to.
- Object orientation allows us to create multiple objects from the same class to store different information.
- To use a Perl class we just `use` the module.
- To create an object we call the constructor method on the class.
- Many objects of the same class can be created.
- To call a method on an object we use the arrow operator.
- Objects are destroyed when they go out of scope.

Chapter 15. Advanced regular expressions

In this chapter...

This chapter builds on the basic regular expressions taught earlier in the course. We will learn how to handle data which consists of multiple lines of text, including how to input data as multiple lines and different ways of performing matches against that data.

Assumed knowledge

You should already be familiar with the following topics:

- Regular expression meta characters
- Quantifiers
- Character classes and alternation
- The `m//` matching function
- The `s///` substitution function
- Matching strings other than `$_` with the =~ matching operator


Patterns and regular expressions are dealt with in depth in chapter 5 (chapter 2, 2nd Ed) of the Camel book, and further information is available in the online Perl documentation by typing **perldoc perlre**.

Capturing matched strings to scalars

Perl provides an easy way to extract matched sections of a regular expression for later use. Any part of a regular expression that is enclosed in parentheses is captured and stored into special variables. The substring that matches first set of parentheses will be stored in `$1`, and the substring that matches the second set of parentheses will be stored in `$2` and so on. There is no limit on the number of parentheses and associated numbered variables that you can use.

```
/(\w)(\w)/;     # matches 2 word characters and stores them in $1, $2
/(\w+)/;        # matches one or more word characters and stores them in $1
```

Parentheses are numbered from left to right by the *opening* parenthesis. The following example should help make this clear:

```
$_ = "fish";
/((\w)(\w))/;   # captures as follows:
                # $1 = "fi", $2 = "f", $3 = "i"

$_ = "1234567890";
/(\d)+/;        # matches each digit and then stores the last digit
                # matched into $1
/(\d+)/;        # captures all of 1234567890
```

Evaluating a regular expression in list context is another way to capture information, with parenthesised sub-expressions being returned as a list. We can use this instead of numbered variables if we like:

```
$_ = "Our server is training.perltraining.com.au.";
my ($full, $host, $domain) = /(([\w-]+)\.([\w.-]+))/;
print "$1\n";                   # prints "training.perltraining.com.au."
print "$full\n";                # prints "training.perltraining.com.au."
print "$2 : $3\n";              # prints "training : perltraining.com.au."
print "$host : $domain\n"       # prints "training : perltraining.com.au."
```

A regular expression that fails to match the given string does not always reset `$1`, `$2` etc. Therefore, if we do not explicitly check that our regular expression worked, we can end up using data from a previous match. This can mean that the following code may cause unexpected surprises:

```
while(<>) {
        # check that we have something that looks like a date in
        # YYYY-MM-DD format.

        if(/(\d{4})-(\d{2})-(\d{2})/) {
                print STDERR "valid date\n";
        }
        next unless $1;

        if($1 >= $recent_year) {
                print RECENT_DATA $_;
        }
        else {
                print OLD_DATA $_;
        }
}
```

If this code encounters a line which doesn't appear to be a valid date, the line may be printed to the same file as the last valid line, rather than being discarded. This could result in lines with dates similar to "1901-3-23" being printed to `RECENT_DATA`, or lines with dates like "2003-1-1" being printed to `OLD_DATA`.

Extended regular expressions

Regular expressions can difficult to follow at times, especially if they're long or complex. Luckily, Perl gives us a way to split a regular expression across multiple lines, and to embed comments into our regular expression. These are known as *extended regular expressions*.

To create a extended regular expression, we use the special `/x` switch. This has the following effects on the match part of an expression:

- Spaces (including tabs and newlines) in the regular expression are ignored.
- Anything after an un-escaped hash (`#`) is ignored, up until the end of line.

Extended regular expressions do not alter the format of the second part in a substition. This must still be written exactly as you wish it to appear.

If you need to include a literal space or hash in an extended expression you can do so by preceeding it with a backslash.

By using extended regular expressions, we can change this:

```
# Parse a line from 'ls -l'
/^([\w-]+)\s+(\d+)\s+(\w+)\s+(\w+)\s+(\d+)\s+(\w+\s+\d+\s+[\d:]+)\s+(.*)$/;
```

into this:

```
# Parse a line from 'ls -l'

/
    ^                           # Start of line.
    ([\w-]+)\s+                 # $1 - File permissions.
    (\d+)\s+                    # $2 - Hard links.
    (\w+)\s+                    # $3 - User
    (\w+)\s+                    # $4 - Group
    (\d+)\s+                    # $5 - File size
    (\w+\s+\d+\s+[\d:]+)\s+     # $6 - Date and time.
    (.*)                        # $7 - Filename.
    $                           # End of line.
/x;
```

As you can see, extended regular expressions can make your code much easier to read, understand, and maintain.

Exercise

For these exercises you may find using the following structure useful:

```
while(<>) {
        chomp;

        my ($origin, $date, $page) = /PATTERN/; # put your regexp here

        ...
}
```

1. Web server access logs typically contain long lines of information, only some of which is of interest at any given time. In the `access-pta.log` file you'll see an example taken from Perl Training Australia's webserver.

   Write a regular expression which captures the request origin, the access date and requested page. Print this out for each access in the file.

You can find an answer to this exercise in `exercises/answers/log-process.pl`.

Advanced Exercise

1. Split tab-separated data into an array then print out each element using a `foreach` loop (an answer's in `exercises/answers/tab-sep.pl`, an example file is in `exercises/tab-sep.txt`).

Greediness

Regular expressions are, by default, "greedy". This means that any regular expression, for instance `.*`, will try to match the biggest thing it possibly can. Greediness is sometimes referred to as "maximal matching".

Greediness is also left to right. Each section in the regular expression will be as greedy as it can while still allowing the whole regular expression to match if possible. For example,

```
$_ = "The cat sat on the mat";

/(c.*t)(.*)(m.*t)/;

print $1;               # prints "cat sat on t"
print $2;               # prints "he "
print $3;               # prints "mat";
```

It is possible in this example for another set of matches to occur. The first expression `c.*t` could have matched `cat` leaving `sat on the` to be matched by the second expression `.*`. However, to do that, we need to stop `c.*t` from being so greedy.

To make a regular expression quantifier not greedy, follow it with a question mark. For example `.*?`. This is sometimes referred to as "minimal matching".

```
$_ = "The fox is in the box.";

/(f.*x)/;             # greedy               -- $1 = "fox is in the box"
/(f.*?x)/;            # not greedy           -- $1 = "fox"

$_ = "abracadabra";

/(a.*a)/              # greedy               -- $1 = "abracadabra"
/(a.*?a)/             # not greedy           -- $1 = "abra"

/(a.*?a)(.*a)/        # first is not greedy  -- $1 = "abra"
                      # second is greedy     -- $2 = "cadabra"

/(a.*a)(.*?a)/        # first is greedy      -- $1 = "abracada"
                      # second is not greedy -- $2 = "bra"

/(a.*?a)(.*?a)/       # first is not greedy  -- $1 = "abra"
                      # second is not greedy -- $2 = "ca"
```

Exercise

1. Write a regular expression that matches the first and last words on a line, and print these out.

More meta characters

Here are some more advanced meta characters, which build on the ones covered earlier.

Table 15-1. More meta characters

Meta character	Meaning
\c*x*	Control character, i.e. **CTRL-*x***
\0*nn*	Octal character represented by *nn*
\x*nn*	Hexadecimal character represented by *nn*
\l	Lowercase next character
\u	Uppercase next character
\L	Lowercase until \E
\U	Uppercase until \E
\Q	Quote (disable) meta characters until \E
\E	End of lowercase/uppercase/quote

```
# search for the C++ computer language:

/C++/        # wrong! regexp engine complains about the plus signs
/C\+\+/      # this works
/\QC++\E/    # this works too

# search for "bell" control characters, eg CTRL-G

/\cG/        # this is one way
/\007/       # this is another -- CTRL-G is octal 07
/\x07/       # here it is as a hex code
```

 Read about all of these and more in **perldoc perlre**.

Working with multi-line strings

Often, you will want to read a file several lines at a time. Consider, for example, a typical Unix fortune cookie file, which is used to generate quotes for the **fortune** command:

```
%
Let’s call it an accidental feature.
        -- Larry Wall
%
Linux: the choice of a GNU generation
%
When you say "I wrote a program that crashed Windows", people just stare at
you blankly and say "Hey, I got those with the system, *for free*".
        -- Linus Torvalds
%
I don’t know why, but first C programs tend to look a lot worse than
first programs in any other language (maybe except for fortran, but then
I suspect all fortran programs look like ‘firsts’)
        -- Olaf Kirch
%
All language designers are arrogant. Goes with the territory...
        -- Larry Wall
%
We all know Linux is great... it does infinite loops in 5 seconds.
        -- Linus Torvalds
%
Some people have told me they don’t think a fat penguin really embodies the
grace of Linux, which just tells me they have never seen a angry penguin
```

```
charging at them in excess of 100mph.  They'd be a lot more careful
about what they say if they had.
        -- Linus Torvalds, announcing Linux v2.0
%
```

The fortune cookies are separated by a line which contains nothing but a percent sign.

To read this file one item at a time, we would need to set the delimiter to something other than the usual `\n` - in this case, we'd need to set it to something like `\n%\n`.

To do this in Perl, we use the special variable `$/`. This is called the input record separator.

```
$/ = "\n%\n";
```

Conveniently enough, setting `$/` to `""` will cause input to occur in "paragraph mode", in which two or more consecutive newlines will be treated as the delimiter. Undefining `$/` will cause the entire file to be slurped in.

```
undef $/;
$_ = <> # whole file now here
```

Changing `$/` doesn't just change how readline (`<>`) works. It also affects the `chomp` function, which always removes the value of `$/` from the end of its argument. The reason we normally think of `chomp` removing newlines is that `$/` is set to newline by default.

It's usually a very good idea to use `local` when changing special variables. For example, we could write:

```
{
        local $/ = "\n%\n";
        $_ = <>;        # first fortune cookie is in $_ now
}
```

to grab the first fortune cookie. By enclosing the code in a block and using local, we restrict the change of `$/` to that block. After the block `$/` is whatever it was before the block (without us having to save it and remember to change it back). This localisation occurs regardless of how you exit the block, and so is particularly useful if you need to alter a special variable for a complex section of code.

Variables changed with `local` are also changed for any functions or subroutines you might call while the `local` is in effect. Unless it was your intention to change a special variable for one or more of the subroutines you call, you should end your block before calling them.

It is a compile-time error to try and declare a special variable using `my`.

Special variables are covered in Chapter 28 of the Camel book, (pages 127 onwards, 2nd Ed). The information can also be found in **perldoc perlvar**.

Since `$/` isn't the easiest name to remember, we can use a longer name by using the **English** module:

```
use English;
```

```
$INPUT_RECORD_SEPARATOR = "\n%\n";      # long name for $/
$RS = "\n%\n";                          # same thing, awk-like
```

The **English** module is documented on page 884 (page 403, 2nd Ed) of the Camel book or in **perldoc English**. You can find out about all of Perl's special variables' English names by reading **perldoc perlvar**.

Exercise

1. In your directory is a file called `exercises/linux.txt` which is a set of Linux-related fortunes, formatted as in the above example. This file contains a great many quotes, including the ones in the example above and many many more. Use multi-line regular expressions to find only those quotes which were uttered by Larry Wall. You might also want to refresh your memory of `chomp()` at this point. (Answer: `exercises/answers/larry.pl`)

Regexp modifiers for multi-line data

Perl has two modifiers for multi-line data. `/s` and `/m`. These can be used to treat the string you're matching against as either a single line or as multiple lines. Their presence changes the behaviour of caret (`^`), dollar (`$`) and dot (`.`).

By default caret matches the start of the string. Dollar matches the end of the string (regardless of newlines). Dot matches anything but a newline character.

With the `/s` modifier, caret and dollar behave the same as in the default case, but dot will match the newline character.

With the `/m` modifier, caret matches the start of any line within the string, dollar matches the end of any line within the string. Dot does not match the newline character.

```
my $string = "This is some text
and some more text
spanning several lines";

if ($string =~ /^and some/m) {                  # this will match
        print "Matched in multi-line mode\n";   # because ^ matches the
}                                               # start of any line
                                                # in the string

if ($string =~ /^and some/s) {                  # this won't match
        print "Matched in single line mode\n";  # because ^ only matches
}                                               # the start of the string.

if($string =~ /^This is some/s) {                # this will match
        print "Matched in single line mode\n";  # (and would have without
}                                               # the /s, or with /m)

if($string =~ /(some.*text)/s) {   # Prints "some text\nand some more text"
        print "$1\n";              # Note that . is matching \n here
}


if($string =~ /(some.*text)/m)  {   # Prints "some text"
        print "$1\n";               # Note that . does not match \n
```

```
}
```

The differences between default, single line, and multi-line mode are set out very succinctly by Jeffrey Friedl in Mastering Regular Expressions (see the Further Reading at the back of these notes for details). The following table is paraphrased from the one on page 236 of that book.

His term "clean multi-line mode" describes one in which each of `^`, `$` and `.` all do what many programmers expect them to do. That is `.` will match newlines as well as all other characters, and `^` and `$` each work on start and end of lines, rather than the start and end of the string.

Table 15-2. Effects of single and multi-line options

Mode	Specified with	^ matches...	$ matches...	Dot matches newline
default	neither `/s` nor `/m`	start of string	end of string	No
single-line	`/s`	start of string	end of string	Yes
multi-line	`/m`	start of line	end of line	No
clean multi-line	both `/m` and `/s`	start of line	end of line	Yes

Modifiers may be clumped at the end of a regular expression. To perform a search using “clean multi-line” irrespective of case your expression might look like this

```
/^the start.*end$/msi
```

and if we had the following strings

```
$string1 = "the start of the day
is the end of the night";

$string2 = "10 athletes waited,
the starting point was ready
how it would end
was anyone’s guess";

$string3 = uc($string2); # same as string 2 but all in uppercase
```

we’d expect the match to succeed with both `$string2` and `$string3` but not with `$string1`.

Back references

Special variables

There are several special variables related to regular expressions. The parenthesised names beside them are their long names if you use the English module.

- `$&` is the matched text (MATCH)
- `$‘` (dollar backtick) is the unmatched text to the left of the matched text (PREMATCH)
- `$’` (dollar forwardtick) is the unmatched text to the right of the matched text (POSTMATCH)
- `$1`, `$2`, `$3`, etc. The text matched by the 1st, 2nd, 3rd, etc sets of parentheses.

All these variables are modified when a match occurs, and can be used in the same way that other scalar variables can be used.

```
my ($match) = m/^(\d+)/;
print $match;

# or alternately...
m/^\d+/;
print $&;

# match the first three words...
m/^(\w+) (\w+) (\w+)/;
print "$1 $2 $3\n";
```

You can also use `$&` and other special variables in substitutions:

```
$string = "It was a dark and stormy night.";
$string =~ s/dark|wet|cold/very $&/;
```

When Perl sees you using PREMATCH (`$‘`), MATCH (`$&`), or POSTMATCH (`$’`), it assumes that you may want to use them again. This means that it has to prepare these variables after every successful pattern match. This can slow a program down because these variables are "prepared" by copying the string you matched against to an internal location.

If the use of those variables make your life much easier, then go ahead and use them. However, if using `$1`, `$2` etc can be used for your task instead, your program will be faster and leaner by using them.

If you want to use parentheses simply for grouping, and don’t want them to set a `$1` style variable, you can use a special kind of *non-capturing* parentheses, which look like `(?: ... )`

```
# this only sets $1 - the first two sets of parentheses are non-capturing
m/^(?:\w+) (?:\w+) (\w+)/;
```

The special variables `$1` and so on can be used in substitutions to include matched text in the replacement expression:

```
# swap first and second words
s/^(\w+) (\w+)/$2 $1/;
```

However, this is no use in a simple match pattern, because `$1` and friends aren’t set until after the match is complete. Something like:

```
my $word = ’t\w+’;
print if m/($word) $1/;
```

... will *not* match "this this" or "that that". Rather, it will match a string containing "this" followed by whatever `$1` was set to by an earlier match.

In order to match "this this" (or "that that") we need to use the special regular expression meta characters `\1`, `\2`, etc. These meta characters refer to parenthesised parts of a match pattern, just as `$1` does, but *within the same match* rather than referring back to the previous match.

```
my $word = 't\w+';
print if m/($word) \1/;
```

Exercises

1. Write a script which swaps the first and the last words on each line.
2. Write a script which looks for doubled terms such as "bang bang" or "quack quack" and prints out all occurrences. This script could be used for finding typographic errors in text. (Answer: `exercises/answers/double.pl`)

Advanced Exercises

1. Make your swapping-words program work with lines that start and end with punctuation characters. (Answer: `exercises/answers/firstlast.pl`)
2. Modify your repeated word script to work across line boundaries (Answer: `exercises/answers/multiline_double.pl`)
3. What about case sensitivity with repeated words?

Chapter summary

- Input data can be split into multi-line strings using the special variable `$/` , also known as `$INPUT_RECORD_SEPARATOR`.
- The `/s` and `/m` modifiers can be used to treat multi-line data as if it were a single line or multiple lines, respectively. This affects the matching of `^` and `$` , as well as whether or not `.` will match a newline.
- The special variables `$&`, `$‘` and `$’` are always set when a successful match occurs.
- `$1`, `$2`, `$3` etc are set after a successful match to the text matched by the first, second, third, etc sets of parentheses in the regular expression. These should only be used *outside* the regular expression itself, as they will not be set until the match has been successful.
- Special non-capturing parentheses `(?:...)` can be used for grouping when you don’t wish to set one of the numbered special variables.
- Special meta characters such as `\1`, `\2` etc may be used *within* the regular expression itself, to refer to text previously matched.

Chapter 16. File I/O

In this chapter...

In this chapter, we learn how to open and interact with files and directories in various ways.

Angle brackets - the line input and globbing operators

We have encountered the line input operator <> in situations such as these:

```
# reading lines from STDIN (or from files on the command line)
while (<>) {
        # Process the line of input in $_
}

# reading a single line of user input from STDIN
my $input = <STDIN>;

# reading all lines from STDIN into an array
my @input = <STDIN>;
```

 The line input operator is discussed in-depth on page 81 (page 53, 2nd Ed) of the Camel book. You can read about the closely-related `readline` function using **perldoc -f readline**.

- In scalar context, the line input operator yields the next line of the file referenced by the filehandle given.
- In list context, the line input operator yields all remaining lines of the file referenced by the filehandle. (Be careful when using this as you may use up all your memory if the file is large).
- The default filehandle is `STDIN`, or any files listed on the command line of the Perl script (eg **myscript.pl file1 file2 file3**).

The *globbing* operator looks the same as the line input operator, but is really quite different.

 The filename globbing operator is documented on page 83 (page 55, 2nd Ed) of the Camel book. You can also read about it with **perldoc perlop**.

If the angle brackets have anything in them other than a filehandle or nothing, it will work as a globbing operator and whatever is between the angle brackets will be treated as a filename wildcard. For instance:

```
my @files = <*.txt>;
```

The filename glob `*.txt` is matched against files in the current directory, then either they are returned as a list (in list context, as above) or one scalar at a time (in scalar context).

Perl's globs operate the same way as they do in the UNIX C-shell. Don't worry if you don't know C-shell, the basic pattern matching operators (such as `*` and `?`) have the same behaviour as just about any other shell that you may have used.

If you get a list of files this way, you can then open them in turn and read from them.

```
while (<*.txt>) {
        open (FILEHANDLE, "< $_") or die ("Can't open $_: $!");

        # Read from the file

        close FILEHANDLE;
}
```

The `glob()` function behaves in a very similar manner to the angle bracket globbing operator.

```
my @files = glob("*.txt");

foreach (glob("*.txt")) {
        # Process the filename in $_
}
```

The `glob()` is considered much cleaner and better to use than the angle-brackets globbing operator.

 Like all functions, you can read more about `glob` using `perldoc -f glob`.

Exercises

1. Use the file globbing function or operator to find all Perl scripts in your current directory and print out their names (assuming they are named in the form `*.pl`) (Answer: `exercises/answers/findscripts.pl`)
2. Use the line input operator to accept and print input from the user on a line-by-line basis.
3. Modify your previous script to use a `while` loop to get user input repeatedly, until they type "Q" (or "q" - check out the `lc()` and `uc()` functions by using `perldoc -f uc` and `perldoc -f lc`) (Answer: `exercises/answers/userinput.pl`)

Advanced exercises

1. Use the above example of globbing to print out all the Perl scripts one after the other. You will need to use the `open()` function to read from each file in turn. (Answer: `exercises/answers/printscripts.pl`)

open() and friends - the gory details

Opening a file for reading, writing or appending

The `open()` function is documented on pages 747-755 (pages 191-195, 2nd Ed) of the Camel book, and also in **perldoc -f open**.

The `open()` function is used to open a file for reading or writing (or both, or as a pipe - more on that later).

In brief, Perl uses the same characters as shell does for file operations. That is:

- < says to open the file for reading
- > says to open the file for writing
- >> says to open the file for appending.

In a typical situation, we might use `open()` to open and read from a file:

```
open(LOGFILE, "< /var/log/httpd/access.log");
```

Note that the < (less than) character used to indicate reading is assumed; we could equally well have said

```
open (LOGFILE, "/var/log/httpd/access.log");
```

Although reading mode is assumed when opening files, it is still always a good idea to explicitly open your files for reading by using the < character. Doing so protects you from the case where your filename has odd characters in it, such as <, > and | which all mean special things to `open`.

Watch out when using > to open files for writing. Like shell, using > will *clobber* any contents of your file. This is because > truncates the file when it is opened. So even if you don't write anything to the file, the original contents will be lost upon opening.

Using > or >> will cause the files to spring into existence if they do not already exist, so you don't have to worry about how to create them before writing.

If you need more control over how you open your files, check out the `sysopen` function by using **perldoc -f sysopen**. Using `sysopen` is especially important if you're running with elevated privileges, as it can help protect against dangerous race conditions. You can read more about that on pages 571-573 in the Camel book (3rd Ed only).

You should *always* check for failure of an `open()` statement:

```
open(LOGFILE, "< /var/log/httpd/access.log")
        or die "Can't open /var/log/httpd/access.log: $!";
```

`die` is a Perl function which takes an error message and terminates the program displaying that message to the user. In this example, the die statement (which is always true) is executed only if the open statement does not return true, that is, if there was an error in opening the file.

Attempting to read from an unopened file may cause unexpected results.

`$!` is the special variable which contains the error message produced by the last system interaction. It is documented in on page 669 (page 134, 2nd Ed) of the Camel book and also in `perldoc perlvar`.

Perl tries to be helpful when dying on errors and will append the appropriate filename and line number of your script to the end of the die message, with a newline. If you don't want this behaviour, end the die message with a newline (`\n`) character. For example:

```
# The following provides an error with file and line-number:
open(LOGFILE, "< $file") or die "Cannot open $file: $!";

# Here the file and line-number are omitted.
open(LOGFILE, "< $file") or die "Cannot open $file: $!\n";
```

Make sure you don't do this by accident, and miss out on this important information.

Once a file is opened for reading or writing, we can use the filehandle we specified (in this case `LOGFILE`) for a variety of useful purposes:

```
open(LOGFILE, "< /var/log/httpd/access.log")
        or die "Can't open /var/log/httpd/access.log: $!";

# use the filehandle in the in the <> line input operator...
while (<LOGFILE>) {
        print if /perltraining.com.au/;
}

close LOGFILE;

# open a new logfile for appending
open(SCRIPTLOG, ">> myscript.log") or die "Can't open myscript.log: $!";

# print() takes an optional filehandle argument - defaults to STDOUT
print SCRIPTLOG "Opened logfile successfully.\n";

close SCRIPTLOG;
```

Note that you should always close a filehandle when you're finished with it (even though any open filehandles will be automatically closed when your script exits).

Under Perl version 5.6.0 and above, you can provide a scalar as the first argument to the `open` function. This means that your filehandles can have scope, and makes it easier to pass them to subroutines and put into structures such as hashes and arrays.

```
my $fh;
open($fh,"< /path/to/file") or die "...";
```

In versions before 5.6.0 you can do the same thing by using the `FileHandle` module, but you need to declare your intentions first:

```
use FileHandle;

my $fh = FileHandle->new;     # $fh is now a FileHandle object.
open ($fh,"< /path/to/file") or die "...";
```

You use scalar filehandles the same way as you use regular ones:

```
while(<$fh>) {
        # do something with each line of the file
}

# print to open filehandle:
print $out_fh "Today is a good day!";

# Or (to make the filehandle stand out more)
print {$out_fh} "Today is a good day!";
```

Using the `FileHandle` module also works in Perl 5.6.0 and above, so if compatibility with older versions of Perl is important to you, you should use the `FileHandle` module for scalar filehandles.

For more information see `perldoc FileHandle` and pages 895-898 (page 442-444, 2nd Ed) in the Camel book.

Be careful when trying to open a file whose name contains characters that might have special meaning to `open()`, in particular those that end with `|` (pipe), or begin with `>` or `<`, as these may result in `open()` not doing what you expect. Leading and trailing spaces are also ignored.

Under Perl 5.6.0 and above, a three-argument version of `open()` exists. This version of `open()` treats the filename literally, including special characters and spaces. You use it like this:

```
my $filename = "filename ending with spaces   ";

open(FILE, "<", $filename)
        or die "Failed to open file: $filename for reading: $!";

while(<FILE>) {
        # Process the line of input in $_
}
```

You can still use scalar filehandles with this version of open.

The three argument version of `open` is much safer than the two-argument version, especially if you're dealing with untrusted user input, as no special interpretation is done on the filename. It's described with the rest of the `open` documentation.

For a safe file open for those who can't upgrade to Perl 5.6, have a look at `sysopen`. Information about sysopen can be found in **perldoc -f sysopen** and pages 808-810 (pages 194, 2nd Ed) of the Camel book.

Exercises

1. Write a script which opens a file for reading. Use a `while` loop to print out each line of the file.
2. Use the above script to open a Perl script. Use a regular expression to print out only those lines not beginning with a hash character (i.e. non-comment lines). (Answer: `exercises/answers/delcomments.pl`)
3. Create a new script which opens a file for writing. Write out the numbers 1 to 100 into this file. (Hint: the numbers 1 to 100 can be generated by using the `..` operator eg: `foreach my $value (1..100) {}`) (Answer: `exercises/answers/100count.pl`)
4. Create a new script which opens a logfile for appending. Create a `while` loop which accepts input from STDIN and appends each line of input to the logfile. (Answer: `exercises/answers/logfile.pl`)
5. Create a script which opens two files, reads input from the first, and writes it out to the second. (Answer: `exercises/answers/readwrite.pl`)

Reading directories

In addition to being able to open files, it's also possible to open directories using the `opendir()` function. Once a directory is open, you can read filehandles from it using the `readdir()` function.

To read the contents of files in the directory, you still need to open each one using the `open()` function.

`opendir()` is documented on page 755 (page 195, 2nd Ed) of the Camel book. `readdir()` is on page 770 (page 202, 2nd Ed). Don't forget that function help is also available by typing **perldoc -f opendir** or **perldoc -f readdir**

```
# $ENV{HOME} stores the home directory on Unix platforms, use
# $ENV{HOMEPATH} for MS Windows
opendir(HOMEDIR, $ENV{HOME}) or die "Can't read dir $ENV{HOME}: $!";

my @files = readdir(HOMEDIR);

closedir HOMEDIR;

foreach (@files) {
        # Skip over directories and non-plain files (eg devices)
        next unless -f "$ENV{HOME}/$_";

        open(THISFILE, "< $ENV{HOME}/$_")
                or die "Can't open file $ENV{HOME}/$_: $!";

        # Read from the file...

        close THISFILE;
}
```

The HOMEDIR in the previous example is a *directory handle* not a filehandle, even though they look the same. Attempting to use a directory handle as a filehandle (or the opposite) will result in an error.

Under Perl 5.6.1 and above you can provide a scalar as the argument to `readdir`. This allows you to have scalar directory handles which have scope and makes it easier for you to pass them to subroutines or include them in hashes and arrays.

```
my $homedir;
opendir($homedir, $ENV{HOME}) or die "Can't read dir $ENV{HOME}: $!";
my @files = readdir($homedir);
```

glob and readdir

There are some major differences between `glob()` and `readdir()`. `glob()` is not as fast but gives you flexibility over which filenames you get back: `glob("*.c")` for example, returns only files with the ".c" extension. `glob()` also gives you back filenames in asciibetical order, whereas `readdir` gives you back the files in whatever order they're stored in the internal representation of your system.

`glob("some/path/*")` will return filenames with path intact whereas readdir will return only the filenames of the files in the directory.

The last difference between these is their behaviour with "." files. For example ".bashrc". `glob("*")` will not return these files (although `glob(".*")` will), whereas `readdir()` will always return "." files.

Table 16-1. Differences between glob and readdir

glob	readdir
Slower	Faster
Allows you to filter filenames	Gives you all filenames
Returns files in asciibetical order	Returns files in file-system order
Returns filename with path intact	Returns filename only
Does not return dot files when called as `glob("*")`	Returns all filenames

rewinddir

We can rewind the current position of the directory handle back to the beginning by using the `rewinddir` function.

```
opendir(HOMEDIR, $ENV{HOME}) or die "Can't read dir $ENV{HOME}: $!";

foreach my $file (readdir(HOMEDIR)) {
        next unless -f "$ENV{HOME}/$_";
        open(THISFILE, "< $ENV{HOME}/$_")
                or die "Can't open file $ENV{HOME}/$_: $!");
        # Read from the file...
}
```

```
rewinddir HOMEDIR;
# Now we can read through our directory contents again...
# if we wish to.
```

`rewinddir` does not refresh the directory listing when it rewinds. To see whether the directory listing has changed since your program started you'll have to close the directory and reopen it.

You can find more about `rewinddir` by reading **perldoc -f rewinddir** or page 777 (page 208, 2nd Ed) in the Camel book.

Changing directories

The function `chdir` allows you to change your program's working directory. All relative file access from that point on will use the new directory. Note that this does *not* change the working directory of the calling process.

```
# Archive log files
my $tar = "/bin/tar";
my $date = "01-01-00";
my $directory = "/var/log/apache/";

unless( chdir($directory) ) {
        die "Failed to chdir to $directory: $!";
}

# Get all files in this directory
my @files = glob("*.log*");
system("$tar -czf weblogs.$date.tgz @files") if @files;

# We learn how to check that system worked later in the course.
```

You can find more about `chdir` by reading **perldoc -f chdir** or page 688 (page 148, 2nd Ed) in the Camel book.

Exercises

1. Use `opendir()` and `readdir()` to obtain a list of files in a directory. What order are they in?
2. Use the `sort()` function to sort the list of files asciibetically (Answer: `exercises/answers/dirlist.pl`)

Changing file contents

When manipulating files, we may wish to change their contents. A flexible way of reading and writing a file is to import the file into an array, manipulate the array, then output each element again.

It is important to ensure that should anything go wrong we don't lose our original data. As a result, it's considered *best-practice* to write our data out to a temporary file and them move that over the input file after everything has been successful.

```
# a program that reads in a file and writes the lines in sorted order.
use File::Copy;

open(INFILE, "< file.txt") or die "Can't open file.txt for input: $!";
my @lines = <INFILE>;   # Slurps all the lines into @lines.
close INFILE;

@lines = sort @lines;

# open temporary file to save our sorted data into
open(OUTFILE, ">file.txt.tmp") or die "Can't open file.txt for output: $!";
foreach my $line (@lines) {
        print OUTFILE $line;
}
close OUTFILE;

# at this point we know that we were successful, so write over the original
# file
move("file.txt.tmp", "file.txt");
```

In fact, since the print function can take a list of values to print, instead of the `foreach` loop above, we could have written:

```
print OUTFILE @lines;
```

The `File::Copy` module makes copying and moving files very easy. To find out more about `File::Copy` read **perldoc File::Copy**.

One thing to watch out for here is memory usage. If you have a ten megabyte file and you read it into an array, it will use at least that much memory as a Perl data structure.

Note that this includes doing things such as:

```
foreach (<FILE>) {
        # Process the line of input in $_
}
```

as this will read the whole file into the list given to `foreach` and then walk over it line by line.

Of course, if you don't need to manipulate all of the lines together (eg sorting) you ought to forgo the reading things into an array and just loop over each line. Continue to ensure, however, that your original data cannot be lost if the program terminates unexpectedly.

```
# removes duplicated lines
open(INFILE, "<file.txt") or die "Can't open file.txt for input: $!";
open(OUTFILE, ">outfile.txt") or die "Can't open outfile.txt for output: $!";
```

```
my $lastline;
while(<INFILE>) {
        print OUTFILE $_ unless ($_ eq $lastline);
        $lastline = $_;
}

close INFILE;
close OUTFILE;
```

Exercises

1. Open a file, reverse its contents (line by line) and write it back to the same filename. For example, "this is a line" would be written as "enil a si siht" (Answer: `exercises/answers/reversefile.pl`)

Opening files for simultaneous read/write

Files can be opened for simultaneous read/write by putting a + in front of the > or < sign. +< is almost always preferable, as +> would overwrite the file before you had a chance to read from it.

Read/write access to a file is not as useful as it sounds --- except under special circumstances (notably when dealing with fixed-length records) you can't usefully write into the middle of the file using this method, only onto the end. The main use for read/write access is to read the contents of a file and then append lines to the end of it.

+< puts you at the start of the file. Note that it won't create a new file if the file you're dealing with does not exist (you'll just get an error that the file doesn't exist). If you start writing before you've reached the end of the file, you will overwrite characters in that file (from that point). Even if you're dealing with fixed-length records and think you know what you're doing, this is often still a bad idea. See below.

+>> initially puts you at the end of the file. It will create a new file if necessary and will not clobber an old one. It allows you to read at any point in the file, but all writes will always go to the end.

Since both print and readline (<>) are buffered, you should *not* use them for editing a file in-place. If you must do so, use the lower level functions such as `sysseek()`, `syswrite()` and `sysread()`. Perl also has a `-i` switch, for more useful in-place modification of files. These concepts are not covered in this course.

For more information about `open` including simultaneous read/write, see **perldoc perlopentut**. Also read pages 747-755 (pages 191-195, 2nd Ed) of the Camel book.

For information about the `-i` option to Perl read **perldoc perlrun** and pages 495-497 (page 332, 2nd Ed) of the Camel book.

For information about Perl's buffering and where it can cause problems read Mark Jason Dominus's excellent article on *Suffering from Buffering* available from his Perl FAQs (http://perl.plover.com/FAQs/Buffering.html).

```
# Example: Reading a file and adding to the end.
# Program that checks to see if $username appears in the file
# adds $username to the end, if not.

my $username = <STDIN>;
chomp $username;

open(FILE, "+< file.txt") or die "Failed to open file.txt for read/write: $!";

my $found;
while(<FILE>) {
        chomp;
                # case insensitive matching
        $found = 1 && last if(lc($_) eq lc($username));
}

# We'll be at the end of our file if $found isn't set
unless($found) {
        print FILE "$username\n";
}
close FILE;
```

Opening pipes

If the filename given to `open()` begins with a pipe symbol (`|`), the filename is interpreted as a command to which output is to be piped, and if the filename ends with a `|`, the filename is to be interpreted as a filename which pipes input to us.

We can use pipes to read information from any process we can execute on our system. Once the command is open, we can read from the resulting filehandle in the same way we would read from any other file. In the example below, we use secure shell (**ssh**) to read a file on a remote machine.

```
#!/usr/bin/perl -w
# This program allows us to read a file from another machine
# using secure shell.  This is most useful if we can login without
# a password (eg, established keys).
use strict;

# Process our command line arguments, and complain if we don't
# have both a host and filename.
my ($host, $file) = @ARGV;
unless ($host and $file) {
        die "Usage: $0 host filename\n";
}

open (SSH, "ssh $host cat $file |") or die "Can't open pipe: $!";

while(<SSH>) {
        # We can process the file in any way we like here.
        # In this particular case, we'll simply print it to
        # our STDOUT.

        print;
}
```

Here's an example which writes to the **sort** command, which is a standard utility on both Windows and Unix systems. Even though Perl has its own **sort** function, the external command is very good at dealing with large amounts of data in a memory-efficient manner.

```
#!/usr/bin/perl -w
use strict;

# Open our external sort command.
open (SORT, "|sort") or die "Can't open pipe: $!";

# Our friends will be printed in sorted order.
foreach my $friend (qw/Jacinta Damian Kirrily Paul/) {
        print SORT "$friend\n";
}
close SORT;
```

If you're interested in reading more about inter-process communication, including pipes, signals, sockets and the like, check out **perldoc perlipc**.

Exercises

1. Modify the second example above (provided for you as `exercises/sort_starter.pl` in your exercises directory) to accept user input and print out the **sort**ed version.
2. Change your script to accept input from a file using `open()` (Answer: `exercises/answers/sort.pl`)
3. If you are using a unix system: change your script to pipe its input through the **strings** command and then **sort**. Now if you specify a file that is not a text file, it will only sort and display printable strings. Try running this over `/usr/bin/perl`. (Answer: `exercises/answers/strings.pl`)

Finding information about files

We can find out various information about files by using file test operators and functions such as `stat()`.

Table 16-2. File test operators

Operator	Meaning
`-e`	File exists.
`-r`	File is readable
`-w`	File is writable
`-x`	File is executable
`-o`	File is owned by you
`-z`	File has zero size.
`-s`	File has nonzero size (returns size).
`-f`	File is a normal file.
`-d`	File is a directory.
`-l`	File is a symbolic link.

Operator	Meaning
-p	File is a named pipe (FIFO), or Filehandle is a pipe.
-S	File is a socket.
-b	File is a block special file.
-c	File is a character special file.
-t	Filehandle is opened to a tty.
-u	File has setuid bit set.
-g	File has setgid bit set.
-k	File has sticky bit set.
-T	File is a text file.
-B	File is a binary file (opposite of -T).
-M	Age of file in days when script started.
-A	Same for access time.
-C	Same for inode change time.

 The file test operators are documented fully in **perldoc -f -x**.

Here's how the file test operators are usually used:

```
#!/usr/bin/perl -w
use strict;

unless (-e "config.txt") {
        die "Config file doesn't exist";
}

# or equivalently...
die "Config file doesn't exist" unless -e "config.txt";
```

The `stat()` function returns similar information for a single file, in list form. `lstat()` can also be used for finding information about a file which is pointed to by a symbolic link. If you've used these functions in C or other languages, then you'll probably find them somewhat familiar in Perl. Check out **perldoc -f stat** to see the format this data is returned in and how to make use of it.

Occasionally it is desirable to perform several tests on the same file at the same time. Perhaps you'd like to check that a file is both readable and writable. It is possible to perform your test like this:

```
if (-r $file && -w $file) {
        # ...
}
```

but that involves two separate tests which both take time. The file might also change between the tests (which is why file tests are almost always a bad idea in security situations).

Perl caches the result of file tests in a special filehandle called _ (underscore). Performing tests on this filehandle can often avoid subsequent system calls, resulting in a slight performance gain.

```
if(-r $file && -w _) {
        # ...
}
```

There are some caveats on when the _ filehandle can be used with certain operators such as `-l` and `-t`. To find out more about these and to learn more about file test operators read **perldoc -f -x**.

The file test operators expect the file you're testing to be in the current working directory. If this is not the case, make sure you prepend a path to the file before doing your test.

Exercises

1. Use the file test operators to print out only files from a directory which are "normal" files, i.e. not directories, symbolic links or other oddities. (Answer: `exercises/answers/normaldirlist.pl`)
2. Write a script to find zero-byte files in a directory. (Answer: `exercises/answers/zerobyte.pl`)
3. Write a script to find the largest file in a directory: `exercises/answers/largestfile.pl`)
4. Write a script which asks a user for a file to open, takes their input from STDIN, checks that the file exists, then prints out the contents of that file. (Answer: `exercises/answers/fileexists.pl`)

Recursing down directories

The **File::Find** module is documented on pages 889-890 (page 439, 2nd Ed) or more fully in **perldoc File::Find**.

The built-in functions described above do not enable you to easily recurse through subdirectories. Luckily, the **File::Find** module is part of the standard library distributed with Perl 5.

`File::Find` emulates Unix's **find** command. It takes as its arguments a subroutine to execute for each file found, and a list of directories to search. Note that to pass a reference to a subroutine we prefix the name of the subroutine with `\&`. In our example below, this is `\&wanted`.

```
#!/usr/bin/perl -w
use strict;
use File::Find;

print "Enter the directory to start searching in: ";
chomp(my $dir = <STDIN>);

# find takes a subroutine reference and the directory to start working from.
find (\&wanted, $dir);
```

```
sub wanted {
        if(/\.pl$/) {                           # See if it's a .pl file
                print "$File::Find::name\n";    # Print the current file name.
        }
}
```

For each file found, certain variables are set.

- `$_` is set to the name of the current file.
- `$File::Find::dir` is set to the directory that contains the file.
- `$File::Find::name` contains the full name of the file, i.e. `$File::Find::dir/$_`.

`File::Find` automatically changes your current working directory to the same as the file you are currently examining. There's rarely a need for `$File::Find::dir`. If all you want to do is process the file regardless of its location on the file system you can simply open the file using the name in `$_`. This behaviour can be turned off, see **perldoc File::Find** for further information.

Exercises

1. Modify the simple script above (in your scripts directory as `exercises/find.pl`) to print out the names of plain text files only (hint: use file test operators)
2. Now use it to print out the contents of each text file. You'll probably want to pipe your output through **less** so that you can see it all. (Answer: `exercises/answers/find.pl`)

File locking

File locking can be achieved using the `flock()` function. This can be used to guard against race conditions or other problems which occur when two (or more) processes want to access the same file at the same time.

`flock()` is documented on page 714 (page 166, 2nd Ed) of the Camel book, or use **perldoc -f flock** to read the online documentation.

`flock` is Perl's portable file-locking mechanism, and works on most operating systems (and produces a fatal error on those which it does not). The locks set by `flock` are advisory only, which means that a process that chooses not to use `flock` can (and will) ignore any locks in place. `flock` can only lock entire files, not individual records. Depending upon your setup, `flock` may or may not work over NFS.

```
# using flock
use Fcntl ':flock';             # import LOCK_* constants

flock(FILEHANDLE, LOCK_EX);     # exclusive (write) access
flock(FILEHANDLE, LOCK_SH);     # shared (read-only) access
flock(FILEHANDLE, LOCK_UN);     # unlock
```

As `flock` only works on *filehandles*, instead of filenames, you have to open the file first *before* you try to lock it. It's important to make sure that you open the file for writing, if you intend to write to it, and that you don't clobber the contents of the file when doing so. This is a good use of `+<`. Closing a locked file releases any locks the process holds upon it. This is good because it means that if your process exits unexpectedly all locks it held are released and other processes may then go forward with their locks.

In the following example, we're locking a file before re-writing it. The exclusive lock stops any other process from holding a lock on the file while we perform our operations.

```
use Fcntl ':flock';             # import LOCK_* constants

# Open file for read and write
open FILE, "+< $file" or die "Failed to open $file: $!";

        # Lock the file for writing (exclusive lock) make sure it worked.
flock(FILE, LOCK_EX) or die "Failed to gain lock on $file: $!";

# At this point we have exclusive access to the file.
# Wipe previous process's details out
truncate(FILE, 0) or die "Failed to truncate $file: $!";

# Write to the file, or perform other operations as needed here...
print FILE $data;

close FILE;             # Closing the file releases the lock as well.
```

`flock` will wait indefinitely until the lock is granted, however it can return early if interrupted by a signal or other event. It's important to ensure that flock returns *true* to be sure that you have the lock you requested. It is possible to make `flock` *non-blocking* as follows:

```
use Fcntl ':flock';             # import LOCK_* constants

flock(FILEHANDLE, LOCK_EX | LOCK_NB);     # non-blocking exclusive lock
flock(FILEHANDLE, LOCK_SH | LOCK_NB);     # non-blocking shared lock
```

All attempts to get a *non-blocking* lock return immediately with either *true* for success (the lock was obtained) or *false* for failure (the lock was not obtained).

For an excellent introduction on using `flock`, the slides from Mark Jason Dominus' *File Locking Tricks and Traps* make excellent reading. They can be found at http://perl.plover.com/yak/flock/.

Handling binary data

If you are opening a file which contains binary data, you probably don't want to read it in a line at a time using `while (<>) { }`, as there's no guarantee that there will be any line breaks in the data, and we'll probably end up with very uneven chunks.

Instead, we can use `read()` to read a certain number of bytes from a file handle. However, before we do that, we should call the `binmode()` function on the filehandle, so that Perl knows that we'll be dealing with a binary file. This means Perl won't try to do any transformations of input based upon the operating system or locale where your program is running.

`binmode()` must be called on the filehandle before any corresponding file I/O. It's best to call it immediately after you open the file.

You can learn more about `read()` by reading page 768 (page 202, 2nd Ed) of the Camel book or **perldoc -f read**.

You can learn more about `binmode()` by reading page 685 (page 147, 2nd Ed) of the Camel book, or **perldoc -f binmode**.

`read()` takes the following arguments:

- The filehandle to read from
- The scalar to put the binary data into
- The number of bytes to read
- The byte offset to start from (defaults to 0)

```
#!/usr/bin/perl -w
use strict;

my $image = "picture.gif";

open (IMAGE, "< $image") or die "Can't open image file: $!";
open (OUT, ">backup/$image") or die "Can't open backup file: $!";

binmode IMAGE;
binmode OUT;

my $buffer;

while (read IMAGE, $buffer, 1024) {
        print OUT $buffer;
}

close IMAGE;
close OUT;
```

Of course, in that particular instance we might have preferred to just copy the file.

Chapter summary

- Angle brackets <> can be used for simple line input. In scalar context, they return the next line; in list context, all remaining lines; the default filehandle is `STDIN` or any files mentioned in the command line (ie `@ARGV`).
- Angle brackets can also be used as a globbing operator if anything other than a filehandle name appears between the angle brackets. In scalar context, returns the next file matching the glob pattern; in list context, returns all remaining matching files.
- The `open()` and `close()` functions can be used to open and close files. Files can be opened for reading, writing, appending, read/write, or as pipes.

- The `opendir()`, `readdir()` and `closedir()` functions can be used to open, read from, and close directories.
- The **File::Find** module can be used to recurse down through directories.
- File test operators or `stat()` can be used to find information about files
- File locking can be achieved using `flock()`
- Binary data can be read using the `read()` function. The `binmode()` function should be used to ensure platform independence when reading binary data.

Chapter 17. System interaction

In this chapter...

In this chapter, we look at different ways to interact with the operating system. In particular, we examine the `system()` function, and the backtick command execution operator. We also look at security and platform-independence issues related to the use of these commands in Perl.

system() and exec()

The `system()` and `exec()` functions both execute system commands.

`system()` forks, executes the commands given in its arguments, waits for them to return, then allows your Perl script to continue. `exec()` does not fork, and exits when it's done. `system()` is by far the more commonly used.

```
% perl -we 'system("/bin/true");  print "Foo\n";'
Foo

% perl -we 'exec("/bin/true");  print "Foo\n";'
Statement unlikely to be reached at -e line 1.
(Maybe you meant system() when you said exec()?)
```

If the command specified by `system()` could not be run the error message will be available via the special variable `$!`. This value is not set if the command can be run but fails during runtime. The return status of the command can be found in the special variable `$?`, which is also the return value of `system()`. This value is a 16-bit status word which needs to be unpacked to be useful, as demonstrated in the example below.

```
system("/bin/true");
if ($?) {                       # A non-zero exit code means failure
                                # -1 means the program didn't even start!
        if($? == -1) {
                print "Failed to run program: $!\n";
        } else {
                print "The exit value was: " .  ($? >> 8) . "\n";
                print "The signal number that terminated the program was: "
                        . ($? & 127) . "\n";
                print "The program dumped core.\n" if $? & 128;
        }
}
```

Just like `open` the traditional form of calling `system` and `exec` have security issues due to shell expansion. For example consider the following code:

```
my $filename;
print "Please give me a file you want to see: ";
$filename = <>;   # lets pretend: $filename="fred; rm -rf /home/pjf;"
chomp($filename);
system("cat $filename");
        # or
exec("cat $filename");
```

In this case, due to shell expansion, the shell will receive the commands:

```
cat fred
rm -rf /home/pjf
```

and if our program had sufficient permissions to delete pjf's home directory, it would.

As a result, there is another, safer, form of `system` and `exec` that bypasses the shell. If you give `system` or `exec` a list it assumes that the first element is the command to execute, and every other element is an argument to that command. These arguments are not passed to the shell, and so shell expansion will not occur. So:

```
system("cat", "*.txt");
       # or
exec("cat", "*.txt");
```

will give the `"*.txt"` filename to `cat` rather than all files with a `.txt` extension. This is essential in cases like the above where the command may be passed in from a user. In this case, if the file "*.txt" does not exist then we'll receive an non-zero return code. Exec fails and returns false only if the command (in this case **cat**) does not exist. If the file does not exist, the user will receive **cat**'s error message.

*nix Exercise

1. Write a script to ask the user for a username on the system, then perform the **finger** command to see information about that user. (Answer: `exercises/answers/finger.pl`)

MS Windows Exercise

1. Write a script to ask the user for a filename on the system. Open the nominated file in Notepad using **system**. (Answer: `exercises/answers/notepad.pl`)

Using backticks

Single quotes can be used to specify a literal string which can be printed, assigned to a variable, et cetera. Double quotes perform interpolation of variables and certain escape sequences such as `\n` to create a string which can also be printed, assigned, etc.

A new set of quotes, called *backticks*, can be used to interpolate variables then run the resultant string as a shell command. The output of that command can then be printed, assigned, and so forth.

Backticks are the backwards-apostrophe character (`) which appears below the tilde (~), next to the number `1` on most keyboards.

Just as the `q()` and `qq()` functions can be used to emulate single and double quotes and save you from having to escape quotes that appear within a string, the equivalent function `qx()` can be used to emulate backticks.

In this course we tend to use `qx()` because it's much harder to confuse `qx()` with plain old single quotes. Using `qx()` also avoids the problem that in some font sets both single quotes and backticks look exactly the same.

Backticks are different to the `system()` command, in that they capture the output of the command they execute, as opposed to passing it through to the user.

When called in a scalar context backticks return the output of the command they execute as a string with possibly embedded newlines. When called in a list context, the output is returned as a list with each separate line of output being a new list element.

```
#!/usr/bin/perl -w
use strict;

# Backticks capture the output of the process they run.  Here,
# we capture the output of the echo command.

my $greeting = qx(echo Hello World);

# $greeting now contains the string "Hello World\n"

# System runs a command without capturing the output, instead it's
# passed straight through.  The following line uses the echo command
# to print a greeting.

system("echo Hello World");
```

The return status of commands called by using backticks can be determined by examining `$?` in the same way as the `system()` example above.

Backticks and the `qx()` function are discussed in the Camel book on page 80 (pages 52 and 41, 2nd Ed) or in **perldoc perlop**.

*nix Exercises

1. Modify your earlier finger program to use backticks instead of `system()` (Answer: `exercises/answers/backtickfinger.pl`)
2. Change it to use `qx()` instead (Answer: `exercises/answers/qxfinger.pl`)
3. The Unix command **whoami** gives your username. Since most shells support backticks, you can type **finger 'whoami'** to finger yourself. Use shell backticks inside your `qx()` statement to do this from within your Perl program. (Answer: `exercises/answers/qxfinger2.pl`)

MS Windows Exercises

1. Modify your earlier program to take a directory path from the user. Use backticks to execute the **DIR** command on that path and list out the files in that directory. (Answer: `exercises/answers/backtickdir.pl`)
2. Change it to use `qx()` instead.

3. Time permitting: reverse sort the directory listing contents.

Platform dependency issues

Note that the examples given above will not work consistently on all operating systems. In particular, the use of `system()` calls or backticks with Unix-specific commands will not work under Windows NT, MacOS, etc. Slightly less obviously, the use of backticks on NT can sometimes fail when the output of a command is sent explicitly to the screen rather than being returned by the backtick operation.

Security considerations

This section is not intended as a comprehensive guide to Perl security, rather it is here to show some of the in-built security features that Perl has available. Even `perldoc perlsec` does not give you the whole picture, it just gives you some hints.

The ability to write secure programs is one that is learnt over many years of experience. It's always a good idea to have someone well rehearsed in security and your programming environment to audit your code in case you have missed anything. As well as training, Perl Training Australia also offers security and privacy auditing services.

Perl Training Australia offers a course in Perl Security that covers many common attacks and mistakes, and how they can be prevented in Perl.

Many of the examples given above can result in major security risks if the commands executed are based on user input. Consider the example of a simple finger program which asked the user who they wanted to finger:

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

use strict;

print "Who do you want to finger? ";
my $username = <STDIN>;
print qx(finger $username);
```

Imagine if the user's input had been `pjf; cat /etc/passwd`, or worse yet, `pjf; rm -rf /`. The system would perform both commands as though they had been entered into the shell one after the other.

A further, not so obvious problem, can be seen when we ask "Which `finger` program are we calling?". If our program caller has changed our `$ENV{PATH}` then it is very possible that it's not the usual system `finger` found in `/usr/bin/`. It could instead be a malicious `finger` program designed to exploit our program's privileges.

Luckily, Perl's `-T` flag can be used to check for unsafe user inputs.

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

Documentation for taint checking can be found by reading the **perldoc perlsec**, or on pages 557-568 (page 356, 2nd Ed) of the Camel book.

`-T` stands for "taint checking". Data input by the user is considered "tainted" and until it has been modified by the script, may not be used to perform shell commands or system interactions of any kind. This includes system interactions such as `open()`, `chmod()`, and any other built-in Perl function which interacts with the operating system.

In versions of Perl prior to 5.8.0 files opened for both reading and writing using "`+<`" were not checked for tainted filenames.

Taint checking will not occur on filenames where the file is only being opened for reading. This is due to historical reasons. Good programming practice would have you untaint these filenames anyway.

The only thing that will clear tainting is referencing substrings from a regexp match. Here's an example.

```
#!/usr/bin/perl -Tw
use strict;

$ENV{PATH} = "/bin:/usr/bin"; # Taint requires we set our path.

print "Who do you want to finger?\n";
my $username = <STDIN>;
chomp($username);

# Check $username to make sure it's clean, then finger.

if ( $username =~ /^(\w{1,8})$/ ) {

       # $1 is the contents of the first set of
       # parentheses in the regexp.

       print qx(finger $1);

} else {
       print "That was not a valid username!\n";
}
```

Make sure you remember to check that the regular expression to untaint your variable succeeded. In the case above we only have one regular expression, so `$1` will either be set by the match or will be undefined. Nevertheless we still explicitly tested the match for success. This means that our code won't break if we add any regular expressions before the code used above.

You can also untaint data by capturing the match in a list context:

```
# Check $username to make sure it's clean
my ($safeuser) = ($username =~ /^(\w{1,8})$/);

# safeuser is now either undefined if the match failed or
# the value of $1 if the match succeeded.
if ($safeuser) {
       print qx(finger $safeuser);
} else {
       print "That was not a valid username!\n";
}
```

Note that you'll have to explicitly set the environment's `PATH` variable (found in `$ENV{PATH}`) to something safe (like `/usr/bin`) as well. This variable affects where the shell looks for other executable programs. `finger` is found in `/usr/bin` on our system.

We have to set a safe value for `$ENV{PATH}` because this value can be changed by the user in their environment before running the Perl script. If the user sets their `PATH` to `/home/pjf/bin` then we'd run the `/home/pjf/bin/finger` command rather than the `/usr/bin/finger` command.

For safety's sake, taint checking in Perl always assumes that the `PATH` environment variable has been tampered with by the user.

If you've been calling your Perl program from the command line with **perl *`program.pl`*** you'll be told that you're turning taint checking on too late, even if you've put it in your shebang line.

This is because Perl wants to know that you want to use taint checking as soon as possible. The way to fix this is to include the `-T` option in your call, so: **perl -T *`program.pl`***.

SETUID scripts automatically run with taint checking turned on for your own protection.

Under Perl 5.8.0 and above, there is also the `-t` switch, which causes tainted operations to generate warnings instead of errors. This is no substitute for real taint checking, but can be useful if you're trying to lock down legacy code and see which areas require attention.

Safe.pm

For greater security when using unknown (and possibly hostile) code, or for writing code which adheres to strict standards about what it's allowed to do, there is the *Safe* module. This module allows the creation of compartments in which Perl code can be evaluated. These compartments allow you to define explicitly what the code run within them may and may not do. For example, you may deny access to the file system so that the code may not read or write to files. Or you may only permit the code to use certain operators such that it may add and subtract but not divide, for example. Attempts by the code to perform forbidden tasks result in a compilation error at compile time and a fatal error at run time.

Note that it is always a good idea to audit code that you receive from a third party before executing it on your machine.

 Learning how to use the *Safe* module is a course in itself. For more information on this module read **perldoc Safe** and pages 576-581 (489-493 2nd Ed) of the Camel book.

Exercise

1. Ask the user for a filename and if that file does not already exist in your directory, open the file and write a short message to it. Turn on taint checking and try running your script. What sort of regular expression could you use to check for valid filenames? (Answer: `exercises/answers/taintfile.pl`)

Chapter summary

- The `system()` function can be used to perform system commands. `$!` is set if any error occurs.
- The backtick operator can be used to perform a system command and return the output. The `qx()` quoting function/operator works similarly to backticks.
- The above methods may not result in platform independent code.
- Data input by users or from elsewhere on the system can cause security problems. Perl's `-T` flag can be used to check for such "tainted" data
- Tainted data can only be untainted by referencing a substring from a pattern match.

Chapter 18. Practical exercises

This chapter provides you with some broader exercises to test your new Perl skills. Each exercise requires you to use a mixture of variables, operators, functions, conditional and looping constructs, and regular expressions.

There are no right or wrong answers. Remember, "There’s More Than One Way To Do It."

1. Write a simple menu system where the user is repeatedly asked to choose a message to display or Q to quit.

    a. Consider case-sensitivity.

    b. Handle errors cleanly.

2. Write a program that gives information about files.

    a. use file test operators.

    b. offer to print the file out if it’s a text file.

    c. how will you cope with files longer than a screenful?

3. Write a "chatterbox" program that holds a conversation with the user by matchings patterns in the user’s input.

Chapter 19. Conclusion

Where to now?

To further extend your knowledge of Perl, you may like to:

- Work through the material included in the appendices of this book.
- Visit the websites in our "Further Reading" section (below).
- Follow some of the URLs given throughout these course notes, especially the ones marked "Readme".
- Install Perl on your home or work computer.
- Practice using Perl from day to day.
- Join a Perl user group such as Perl Mongers (http://www.pm.org/).
- Join an on-line Perl community such as PerlMonks (http://www.perlmonks.org/).
- Extend your knowledge with further Perl Training Australia courses such as:
  - Web Development with Perl
  - Database Programming with Perl
  - Perl Security
  - Object Oriented Perl

  Information about these courses can be found on Perl Training Australia's website (http://www.perltraining.com.au/).

Further reading

Books

- Larry Wall, Tom Christiansen and Jon Orwant, *Programming Perl* (3rd Ed), O'Reilly and Associates, 2000. ISBN 0-596-00027-8
- Tom Christiansen and Nathan Torkington, *The Perl Cookbook*, O'Reilly and Associates, 1998. ISBN 1-56592-243-3.
- Jeffrey Friedl, *Mastering Regular Expressions*, O'Reilly and Associates, 1997. ISBN 1-56592-257-3.
- Joseph N. Hall and Randal L. Schwartz *Effective Perl Programming*, Addison-Wesley, 1997. ISBN 0-20141-975-0.
- Damian Conway, *Perl Best Practices*, O'Reilly and Associates, 2005. ISBN 0-59600-173-8.

Online

- Perl Mongers Perl user groups (http://www.pm.org/)
- PerlMonks online community (http://www.perlmonks.org/)
- Comprehensive Perl Archive Network (http://www.cpan.org)
- The Perl homepage (http://www.perl.com/)
- The Perl Journal (http://www.tpj.com/)
- Perlmonth (http://www.perlmonth.com/) (online journal)
- comp.lang.perl.announce newsgroup
- comp.lang.perl.moderated newsgroup
- comp.lang.perl.misc newsgroup

Appendix A. Advanced Perl Variables

In this chapter...

In this chapter we will explore Perl's variable types a little further. We'll look at hash slices and cool ways to assign values into and from arrays and hashes. But first we'll look at how we can make quoting a little nicer.

Quoting with `qq()` and `q()`

Using double quotes or single quotes when quoting some strings can result in lots of character escaping. Which quotes are best for quoting the following paragraph?

```
Jamie and Peter's mother couldn't drive them to the show.
"How are we going to get there?" Jamie asked.
"We could ride our bikes", Peter suggested.
But Peter's bike had a flat tyre.
```

If we use double quotes it comes out looking like this:

```
print "Jamie and Peter's mother couldn't drive them to the show.
\"How are we going to get there?\" Jamie asked.
\"We could ride our bikes\", Peter suggested.
But Peter's bike had a flat tyre.";
```

but that's just ugly. Single quotes aren't much better:

```
print 'Jamie and Peter\'s mother couldn\'t drive them to the show.
"How are we going to get there?" Jamie asked.
"We could ride our bikes", Peter suggested.
But Peter\'s bike had a flat tyre.';
```

In order to encourage beautiful code that you can be proud of, Perl allows you to pick your own quote operators, when you need to, by providing you with `q()` and `qq()`. `q()` represents single quotes and `qq()` represents double quotes. Note that the same rules apply for each of these quoting styles as for their more common equivalents: `qq()` allows variable interpolation and control character expansion (such as the newline character) whereas `q()` does not. These are often called "pick your own quotes" or "roll your own quotes".

Using pick your your own quotes, quoting the above paragraph becomes easy:

```
qq(Jamie and Peter's mother couldn't drive them to the show.
"How are we going to get there?" Jamie asked.
"We could ride our bikes", Peter suggested.
But Peter's bike had a flat tyre.);
```

You may use any non-whitespace, non-alphanumeric character as your delimiters. Pick one not likely to appear in your string. Note that things that look like they should match up do. So `(` matches `)`, `{` matches `}` and finally < matches >. There are some illustrated below.

```
print q/Jamie said "Using slashes as quoting delimiters is very common."/;
print q(Jamie said "You should always watch your quotes!");
print qq!Jamie said "$these are Paul's favourite quotes". (He was wrong).\n!;
print qq[Jamie said "Perl programs ought to start with #!"\n];
print qq#Jamie said "My favourite regexp is '/[jamie]*/i;'"\n#;
```

If you use matching delimiters around your quoted text Perl will allow you to include those delimiters in your quoted text if they are also paired.

```
print qq(There was a (large) dog in the yard\n);      # This will work
```

If the delimiters within your quoted text are not paired, this will result in errors.

```
print qq< 1 + 4 < 10 >;                               # This will not work
```

The problem with the last example is that Perl assumes that the closing > is paired with the second < and waits to see a later > to close the string.

A different way of quoting strings are HERE documents. These can sometimes be confusing for the reader, and usually pick your own quotes will be clearer. We cover HERE documents here for the sake of completeness, and because they are still very common in older code. If you've done a lot of shell programming you may recognise this construct. HERE documents allow you to quote text up to a certain marking string. For example:

```
print << "END";
I can print any text I want to put here without
fear of "weird" things happening to it.    All
punctuation is fine, unlike roll-your-own quotes,
where you have to pick some kind of punctuation to
delimit it.    Here, we just have to make sure that
the word, up there (next to print) does not appear
in this text, on a line by itself and unquoted.
Otherwise we terminate our text.
END
```

The quoting style used in HERE documents is whatever you quote the terminating word with next to the print statement (in this case double quotes). Using double quotes results in variable interpolation, whereas using single quotes results in no variable interpolation.

Exercises

1. Experiment with using `q()` and `qq()` to print the following string:

   `\/<+c&b^$!a@_#'*'"~{ [()] }~"'*'#_@a!$^b&c+>\/`
   you'll find this string in the file: `exercises/quoteme.pl`

You'll find answers to the above in `exercises/answers/quoted.pl`.

Scalars in assignment

You may find yourself wishing to declare and initialise a number of variables at once:

```
my $start = 0;
my $end = 100;
my $mid = 50;
```

but you don't want to take up three lines to do it in. Perl lets you do the following:

```
my ($start, $end, $mid) = (0, 100, 50);
```

which says create the variables `$start`, `$end` and `$mid` and assign them values from the list on the right dependent on their list position. You'll see this kind of thing all the time. If the list on the right is longer than the list on the left, the extra values are ignored. If the list on the left is longer than the list on the right, the extra variables get no value.

```
my ($a, $b, $c) = (1, 2, 3, 4, 5);   # $a = 1, $b = 2, $c = 3.
                                     # values 4 and 5 are ignored.
my ($d, $e, $f, $g) = (1, 3, 5);     # $d = 1, $e = 3, $f = 5.
                                     # $g gets no value.
```

If the variables are already declared with `my` elsewhere, you can still use the above method to assign to them.

```
($a, $b, $c) = (1, 4, $d);           # $a = 1, $b = 4, $c = $d.
```

In fact, this gives us a very simple way to swap the values of two variables without needing a temporary variable:

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

You'll notice above that in all the examples we've grouped our lists within parentheses. These parentheses are required.

Arrays in assignment

Just as we could assign a list of values to a list of scalars, we can assign elements from arrays to a list of scalars as well. Once again if we provide more values on the right than we provide variables on the left, the extra ones are ignored. If we provide more variables on the left than values on the right, the extra variables are given no value.

```
my ($fruit1, $fruit2, $fruit3) = @fruits;         # assign from array
my ($number1, $number2) = @magic_numbers[-2, -1]; # assign from array slice
my @short = (1,2);
my ($a, $b, $c) = @short;                         # $c gets no value
($a, $b) = @random_scalars;                       # changes $a and $b.
```

Sometimes we would like to make sure that we get enough values in our list to initialise all of our variables. We can do this by supplementing our list with reasonable defaults:

```
my @short = (1, 2);
my ($a, $b, $c, $d, $e, $f) = (@short, 0, 0, 0, 0, 0, 0);
```

this way, even if `@short` is completely empty we know that our variables will all be initialised.

So what happens if you put an array on the left hand side? Well, you end up with an array copy.

```
my @other_fruits = @fruits;        # copies @fruits into @other_fruits
my @small_fruits = @fruits[0..2];  # copies apples, oranges and guavas into
                                   # @small_fruits.
```

What happens if you put two arrays on the left and two on the right? Do you end up with two array copies? Can you use this to swap the contents of two arrays? Unfortunately no.

```
(@a, @b) = (@c, @d);              # Does @a = @c, @b = @d ?   No.
                                  # Instead:
                                  # @a = @c and @d joined together
                                  # @b is made empty

(@a, @b) = (@b, @a);              # Are array contents swapped? No.
                                  # Instead:
                                  # @a becomes @b and @a joined together
                                  # @b is made empty.
```

When two arrays are put together into a list, they are "flattened" and joined together. This is great if you wish to join two arrays together:

```
my @bigger = (@small1, @small2, @small3);         # join 3 arrays together
```

but a bit awkward if you were hoping to swap their contents. To get two array copies or to swap the contents of two arrays, you're going to have to do it the long way.

Hash slices

Hash slices are used less frequently than array slices and are usually considered more confusing. To take a hash slice we do the following:

```
# Our hash
my %people = (
        James => 30,
        Ralph => 5,
        John => 23,
        Jane => 34,
        Maria => 26,
        Bettie => 29
);

# An array (some of the people in %people)
my @friends = qw/Bettie John Ralph/;

# Taking a hash slice on the %people hash using the array @friends to
# give us the keys.
my @ages = @people{@friends};                   # @ages contains: 29, 23, 5

my @ages_b = @people{qw/Bettie John Ralph/;};  # essentially the same as above
```

You'll notice that when we did the hash slice we used an `@` symbol out the front rather than a `%` sign. This isn't a typographical error. The reason we use an `@` sign is because we're expect a list (of values) out. Perl knows that we're looking at the hash called `%people` rather than any array called `@people` because we've used curly braces rather than square brackets.

We can also assign values to a hash slice in the same way we can assign values to a list. This allows us to use hash slices when we wish to see if a number of things exist in an array without traversing the array each time. This is important because if the array is large, searching through all of it multiple times may be infeasible.

```
# The array of things we'd like to test against
my @colours = qw/red green yellow orange/;

# A list of things that might be in @colours or not
my @find = qw/red blue green black/;
```

```
my %colours;                # hashes and arrays can have the same names.
                            # hash slices use curly braces {} and
                            # array slices use square brackets []

@colours{@colours} = ();    # set all values in %colours from the keys in
                            # @colours to have the undefined value (but exist in
                            # the hash).

                            # We now look for @find in %colours rather than
                            # @colours.  This is much faster.
foreach my $colour (@find) {
        if(exists( $colours{$colour} )) {
                print "true  ";
        }
        else {
                print "false  ";
        }
}
```

Exercise

We can use the fact that hash keys are unique to remove duplicates from an array.

1. Taking the list:

   ```
   qw/one one one two three three three four four five five five/;
   ```

   use a hash slice to print out only the unique values. (Don't worry about the order they come out in).

2. Use a hash slice and a foreach loop to print out the unique values of the above list in first-seen order (ie: one two three four five).

Answers for the above questions can be found in `exercises/answers/duplicates.pl`.

Hashes in assignment

Assignment from hashes is a little different to assignment from arrays. If you try the following:

```
my ($month1, $month2) = %monthdays;
```

you won't get the names of two months. When a hash is treated as a list it flattens down into a list of key-value pairs. This means that `$month1` will certainly be the name of a month, but `$month2` will be the number of days in `$month1`.

To get all they keys of a hash we use the `keys` function. If we wanted two of these we can do the following:

```
my ($month1, $month2) = keys %monthdays;
```

To get two values from this hash (which would match the keys we've pulled out above) we use the `values` function.

```
my ($days1, $days2) = values %monthdays;
```

As the `values` function only returns the values inside the hash and we cannot easily determine from a value which key it had, using the `values` function loses information. Usually the values in a hash are accessed through their keys:

```
my $days1 = $monthdays{January};
my $days2 = $monthdays{February};

my ($days1, $days2) = @monthdays{qw/January February/};  # a shorter way
                                                         # if we want a few
```

We can use the fact that hashes flatten into lists when used in list context to join hashes together.

```
my %bigger = (%smaller, %smallest);
```

Note, however, that because each hash key must be unique that this may result in some data loss. When two hash keys clash the earlier one is over written with the later one. In the case above, any keys in `%smaller` that also appear in `%smallest` will get the values in `%smallest`. This is great news if you have a hash of defaults you want to use if any values are missing.

```
my %defaults = (
        name => "John Doe",
        address => "No fixed abode",
        age => "young",
);

my %input = (
        name => "Paul Fenwick",
        address => "c/o Perl Training Australia",
);

%input = (%defaults, %input);       # join two hashes, replacing defaults
                                    # with provided values
                                    # age was missing; gets set to "young"
```

To copy a hash you can just assign its value to the copy hash. However, attempts to perform a double copy in one step or to swap the values of two hashes without a temporary hash result in the same issues as with arrays due to list flattening.

Chapter summary

- Using `q()` and `qq()` allows the programmer to chose quoting characters other than `"` and `'`.
- Perl allows paired delimiters to also appear in the quoted text when using `q()` and `qq()` so long as those characters are also paired.
- Perl allows programmers to initialise scalar variables from lists and to provide less or more values than required if desired.
- You can swap the value of two scalar variables by assigning their values to each other in a list assignment.
- Arrays can be copied by assigning one array to another.
- Arrays flatten to one big list when combined in list context.
- Hash slices allow us to access several values from a hash in one step.
- Hashes can be copied by assigning one hash to another.

Appendix B. Named parameter passing and default arguments

In this chapter...

In this chapter we look at how we can improve our subroutines by using named parameter passing and default arguments. This is commonly used in object oriented Perl programming but is of great use whenever a subroutine needs to take many arguments, or when it is of use to allow more than one argument to be optional.

Named parameter passing

As you will have seen, Perl expects to receive scalar values as subroutine arguments. This doesn't mean that you can't pass in an array or hash, it just means that the array or hash will be flattened into a list of scalars. We can reconstruct that list of scalars into an array or hash so long as it was the final argument passed into the subroutine.

Most programming languages, including Perl, pass their arguments *by position*. So when a function is called like this:

```
interests("Paul","Perl","Buffy");
```

the `interests()` function gets its arguments in the same order in which they were passed (in this case, `@_` is `("Paul","Perl","Buffy")`). For functions which take a few arguments, positional parameter passing is succinct and effective.

Positional parameter passing is not without its faults, though. If you wish to have optional arguments, they can only exist in the end position(s). If we want to take extra arguments, they need to be placed at the end, or we need to change every call to the function in question, or perhaps write a new function which appropriately rearranges the arguments and then calls the original. That's not particularly elegant. As such, positional passing results in a subroutine that has a very rigid interface, it's not possible for us to change it easily. Furthermore, if we need to pass in a long list of arguments, it's very easy for a programmer to get the ordering wrong.

Named parameter passing takes an entirely different approach. With named parameters, order does not matter at all. Instead, each parameter is given a name. Our `interests()` function above would be called thus:

```
interests(name => "Paul", language => "Perl", favourite_show => "Buffy");
```

That's a lot more keystrokes, but we gain a lot in return. It's immediately obvious to the reader the purpose of each parameter, and the programmer doesn't need to remember the order in which parameters should be passed. Better yet, it's both flexible and expandable. We can let any parameter be optional, not just the last ones that we pass, and we can add new parameters at any time without the need to change existing code.

The difference between positional and named parameters is that the named parameters are read into a hash. Arguments can then be fetched from that hash by name.

```
interests(name => "Paul", language => "Perl", favourite_show => "Buffy");
```

```
sub interests {
        my (%args) = @_;

        my $name = $args{name} || "Bob the Builder";
        my $language = $args{language} || "none that we know";
        my $favourite_show = $args{favourite_show} || "the ABC News";

        print "${name}'s primary language is $language.  " .
              "$name spends their free time watching $favourite_show\n";
}
```

Calling a subroutine or method with named parameters does not mean we're passing in an anonymous hash. We're passing in a list of `name => value` pairs. If we wanted to pass in an anonymous hash we'd enclose the name-value pairs in curly braces `{}` and receive a hash reference as one of our arguments in the subroutine.

Some modules handle arguments this way, such as the `CGI` module, although `CGI` also accepts `name => value` pairs in many cases.

It is important to notice the distinction here.

Default arguments

Using named parameters, it's very easy for us to use defaults by merging our hash of arguments with our hash of arguments, like this:

```
my %defaults = ( pager => "/usr/bin/less", editor => "/usr/bin/vim" );

sub set_editing_tools {
        my (%args) = @_;

        # Here we join our arguments with our defaults.    Since when
        # building a hash it's only the last occurrence of a key that
        # matters, our arguments will override our defaults.
        %args = (%defaults, %args);

        # print out the pager:
        print "The new text pager is: $args{pager}\n";

        # print out the editor:
        print "The new text editor is: $args{editor}\n";
}
```

Exercises

1. Rewrite the `interests` subroutine above to use a hash of default arguments rather than assigning individual defaults.

Subroutine declaration and prototypes

Many programming languages allow or require you to predeclare your subroutines/functions. These declarations, also called prototypes, tell the compiler what types of arguments the subroutine is expecting. Should the subroutine then be passed too few, too many or the wrong kind of arguments; a compile-time error is generated and the program does not run.

While prototypes in Perl do exist, they are not the same as the above mentioned function declarations. Prototypes allow developers to write subroutines which mimic Perl's built-in functions, but they don't work the same was as they do in other languages. When used with regular subroutines, the consequences can be surprising and difficult to understand.

It is recommended that you avoid using Perl's subroutines prototypes.


Should you have a requirement to validate your subroutine parameters the `Params::Validate` module, available from CPAN, will do all that you want and more.

Chapter summary

- Parameters in Perl are usually passed "by position".
- Positional parameter passing makes having independent optional arguments or extra arguments difficult.
- Using positional parameter passing requires the programmer to remember or look up the parameter order when dealing with subroutines that take many arguments.
- Named parameter passing makes independent optional arguments and extra arguments easy.
- Named parameter passing allows the programmer to list the arguments in an easy to understand and change manner.
- Using named parameter passing, it becomes very easy to create default values for parameters.

Appendix C. Complex data structures

References are most often used to create complex data structures. Since references are scalars, they can be used as values in both hashes and arrays. This makes it possible to create both deep and complex multi-dimensional data structures. We'll cover some of these in further detail in this chapter.


Complex data structures are covered in detail in chapter 9 (chapter 4, 2nd Ed) of the Camel book.

Arrays of arrays

The simplest kind of nested data structure is the two-dimensional array or matrix. It's easy to understand, use and expand.

Creating and accessing a two-dimensional array

To create a two dimensional array, use anonymous array references:

```
my @AoA = (
        [qw(apple orange pear banana)],
        [qw(mouse rat hamster gerbil rabbit)],
        [qw(camel llama panther sheep)],
);

print $AoA[1]->[3];       # prints "gerbil"
```


The arrow is optional between brackets or braces so the above access could equally well have been written:

```
print $AoA[1][3];
```

Adding to your two-dimensional array

There are several ways you can add things to your two-dimensional array. These also apply to three and four and five and n-dimensional arrays. You can push an anonymous array into your array:

```
push @AoA, [qw/lions tigers bears/];
```

or assign it manually:

```
$AoA[5] = [qw/fish chips vinegar salt pepper-pop/];
```

You can also add items into your arrays manually:

```
$AoA[0][5] = "mango";
```

or by pushing:

```
push @{$AoA[0]}, "grapefruit";
```

You're probably wondering about why we needed the curly braces in our last example. This is because we want to tell Perl that we're looking at the element `$AoA[0]` and asking it to deference that into an array. When we write `@$AoA[0]` Perl interprets that as `@{$AoA}[0]` which assumes that `$AoA` is a reference to an array we're trying to take an array slice on it. It's usually a good idea to use curly braces around the element you're dereferencing to save everyone from this confusion.

Printing out your two-dimensional array

Printing out a single element from your two-dimensional array is easy:

```
print $AoA[1][2];           # prints "hamster"
```

however, if you wish to print our your data structure, you can't just do this:

```
print @AoA;
```

as what you'll get is something like this:

```
ARRAY(0x80f606c)ARRAY(0x810019c)ARRAY(0x81001f0)
```

which are stringified references. Instead you'll have to create a loop to print out your array:

```
foreach my $list (@AoA) {
        print "@$list";
}
```

Hashes of arrays

Arrays of arrays have their uses, but require you to remember the row number for each separate list. Hashes of arrays allow you to associate information with each list so that you can look up each array from a key.

Creating and accessing a hash of arrays

To create a hash of arrays create a hash whose keys are anonymous arrays:

```
my %HoA = (
        fruits =>  [qw(apple orange pear banana)],
        rodents => [qw(mouse rat hamster gerbil rabbit)],
        books =>   [qw(camel llama panther sheep)],
);

print $HoA{rodents}[3];         # prints "gerbil"
```

Adding to your hash of arrays

Adding things to your hash of arrays is easy. To add a new row, just assign an anonymous array to your hash:

```
$HoA{oh_my} =  [qw/lions tigers bears/];
```

To add a single element to an array, either add it in place or push it on the end:

```
$HoA{fruits}[4] = "grapefruit";
push @{$HoA{fruits}}, "mango";
```

Once again you'll notice that we needed an extra set of curly braces to make it clear to Perl that we wanted `$HoA{fruits}` dereferenced to an array.

Printing out your hash of arrays

Printing out a single element from your hash of arrays is easy:

```
print $HoA{fruits}[2];          # prints "pear"
```

Printing out all the element once again requires a loop:

```
foreach my $key (keys %HoA) {
        print "$key => @{$HoA{$key}}\n";
}
```

Arrays of hashes

Arrays of hashes are particularly common when you have number of ordered records that you wish to process sequentially, and each record consists of key-value pairs.

Creating and accessing an array of hashes

To create an array of hashes create an array whose values are anonymous hashes:

```
my @AoH = (
        {
                name => "John",
                age => 31,
        },
        {
                name => "Mary",
                age => 23,
        },
        {
                name => "Paul",
                age => 27,
        },
);

print $AoH[2]{name};            # prints "Paul"
```

Adding to your array of hashes

To add a new hash to your array, add it manually or push it on the end. To add an element to every hash use a loop:

```
$AoH[3] = {                             # adding a new hash manually
                name => "Jacinta",
                age => 26,
          };

push @AoH, {                            # pushing a new hash on to the end
                name => "Judy",
                age => 47
           };

$AoH[0]{favourite_colour} = "blue";     # adding an element to one hash

foreach my $hashref (@AoH) {            # adding an element to every hash
        $hash->{language} = "Perl";
}
```

Printing out your array of hashes

To print a array of hashes we need two loops. One to loop over every element of the array and a second to loop over the keys in the hash:

```
foreach my $hashref (@AoH) {
        foreach $key (keys %$hashref) {
                print "$key => $hashref->{$key}\n";
        }
}
```

Hashes of hashes

Hashes of hashes are an extremely common sight in Perl programs. Hashes of hashes allow you to have a number of records indexed by name, and for each record to contain sub-records. As hash lookups are very fast, accessing data from the structures is also very fast.

Creating and accessing a hash of hashes

To create a hash of hashes, assign anonymous hashes as your hash values:

```
my %HoH = (
        Jacinta => {
                        age => 26,
                        favourite_colour => "blue",
                        sport => "swimming",
                        language => "Perl",
                   },
        Paul => {
                        age => 27,
                        favourite_colour => "green",
                        sport => "cycling",
                        language => "Perl",
                },
```

```
        Ralph => {
                        age => 7,
                        favourite_colour=> "yellow",
                        sport => "little athletics",
                        language => "English"
                 },
);

print $HoH{Ralph}{sport};       # prints "little athletics"
```

Adding to your hash of hashes

```
$HoH{Ralph}{favourite_food} = "Tomato sauce";   # adding to Ralph's hash

$HoH{Tina} = {                                  # adding a new person hash
                age => 19,
                favourite_colour => "black",
                sport => "tai chi",
};
```

Printing out your hash of hashes

Once again, to print out a hash of hashes we'll need two loops, one for each key of the primary hash and the second for each key of the inner hash.

```
foreach my $person ( keys %HoH ) {
        print "We know this about $person:\n";
        foreach $key ( keys %{ $HoH{$person} } ) {
                print "${person}'s $key is $HoH{$person}{$key}\n";
        }
        print "\n";
}
```

More complex structures

Armed with an understanding of the nested data structures we've just covered you should be able to create the best data structure for what you need. Perhaps you need a hash of hashes but where some of your values are arrays. This should pose no problems. Perhaps you want an array of hashes of arrays? This too should be easy.

Appendix D. More functions

The grep() function

The `grep()` function is used to search a list for elements which match a certain regexp pattern. It takes two arguments - a pattern and a list - and returns a list of the elements which match the pattern.

 The `grep()` function is on page 730 (page 178, 2nd Ed) of your Camel book.

```
# trivially check for valid email addresses
my @valid_email_addresses = grep /\@/, @email_addresses;
```

The `grep()` function temporarily assigns each element of the list to `$_` then performs matches on it.

There are many more complicated uses for the grep function. For instance, instead of a pattern you can supply an entire block which is to be used to process the elements of the list.

```
my @long_words = grep { (length($_) > 8); } @words;
```

`grep()` doesn't require a comma between its arguments if you are using a block as the first argument, but does require one if you're just using an expression. Have a look at the documentation for this function to see how this is described.

Exercises

1. Use `grep()` to return a list of elements which contain numbers (Answer: `exercises/answers/grepnumber.pl`)
2. Use `grep()` to return a list of elements which are
   a. keys to a hash (Answer: `exercises/answers/grepkeys.pl`)
   b. readable files (Answer: `exercises/answers/grepfiles.pl`)

The map() function

The `map()` function can be used to perform an action on each member of a list and return the results as a list.

```
my @lowercase = map lc, @words;
my @doubled = map { $_ * 2 } @numbers;
```

`map()` is often a quicker way to achieve what would otherwise be done by iterating through the list with `foreach`.

```
foreach (@words) {
        push (@lowercase, lc($_);
}
```

Like `grep()`, it doesn't require a comma between its arguments if you are using a block as the first argument, but does require one if you're just using an expression.

Exercises

1. Create an array of numbers. Use `map()` to calculate the square of each number. Print out the results.

Appendix E. Unix cheat sheet

A brief run-down for those whose Unix skills are rusty:

Table E-1. Simple Unix commands

Action	Command
Change to home directory	**cd**
Change to *directory*	**cd** ***directory***
Change to directory above current directory	**cd ..**
Show current directory	**pwd**
Directory listing	**ls**
Wide directory listing, showing hidden files	**ls -al**
Showing file permissions	**ls -al**
Making a file executable	**chmod +x** ***filename***
Printing a long file a screenful at a time	**more** ***filename*** or **less** ***filename***
Getting help for *command*	**man** ***command***

Appendix F. Editor cheat sheet

This summary is laid out as follows:

Table F-1. Layout of editor cheat sheets

Running	Recommended command line for starting it.
Using	Really basic howto. This is not even an attempt at a detailed howto.
Exiting	How to quit.
Gotchas	Oddities to watch for.
Help	How to get help.

vi (or vim)

Running

```
% vi filename

      or

% vim filename (where available)
```

Using

- `i` to enter insert mode, then type text, press **ESC** to leave insert mode.
- `x` to delete character below cursor.
- `dd` to delete the current line
- Cursor keys should move the cursor while *not* in insert mode.
- If not, try `hjkl`, `h` = left, `l` = right, `j` = down, `k` = up.
- `/`, then a string, then **ENTER** to search for text.
- `:w` then **ENTER** to save.

Exiting

- Press **ESC** if necessary to leave insert mode.
- `:q` then **ENTER** to exit.
- `:q!` **ENTER** to exit without saving.
- `:wq` to exit with save.

Gotchas

vi has an insert mode and a command mode. Text entry only works in insert mode, and cursor motion only works in command mode. If you get confused about what mode you are in, pressing **ESC** twice is guaranteed to get you back to command mode (from where you press i to insert text, etc).

Help

`:help` **ENTER** might work. If not, then see the manpage.

nano (pico clone)

Running

```
% nano -w filename
```

Using

- Cursor keys should work to move the cursor.
- Type to insert text under the cursor.
- The menu bar has ^X commands listed. This means hold down **CTRL** and press the letter involved, eg **CTRL-W** to search for text.
- **CTRL-O** to save.

Exiting

Follow the menu bar, if you are in the midst of a command. Use **CTRL-X** from the main menu.

Gotchas

Line wraps are automatically inserted unless the -w flag is given on the command line. This often causes problems when strings are wrapped in the middle of code and similar.

Help

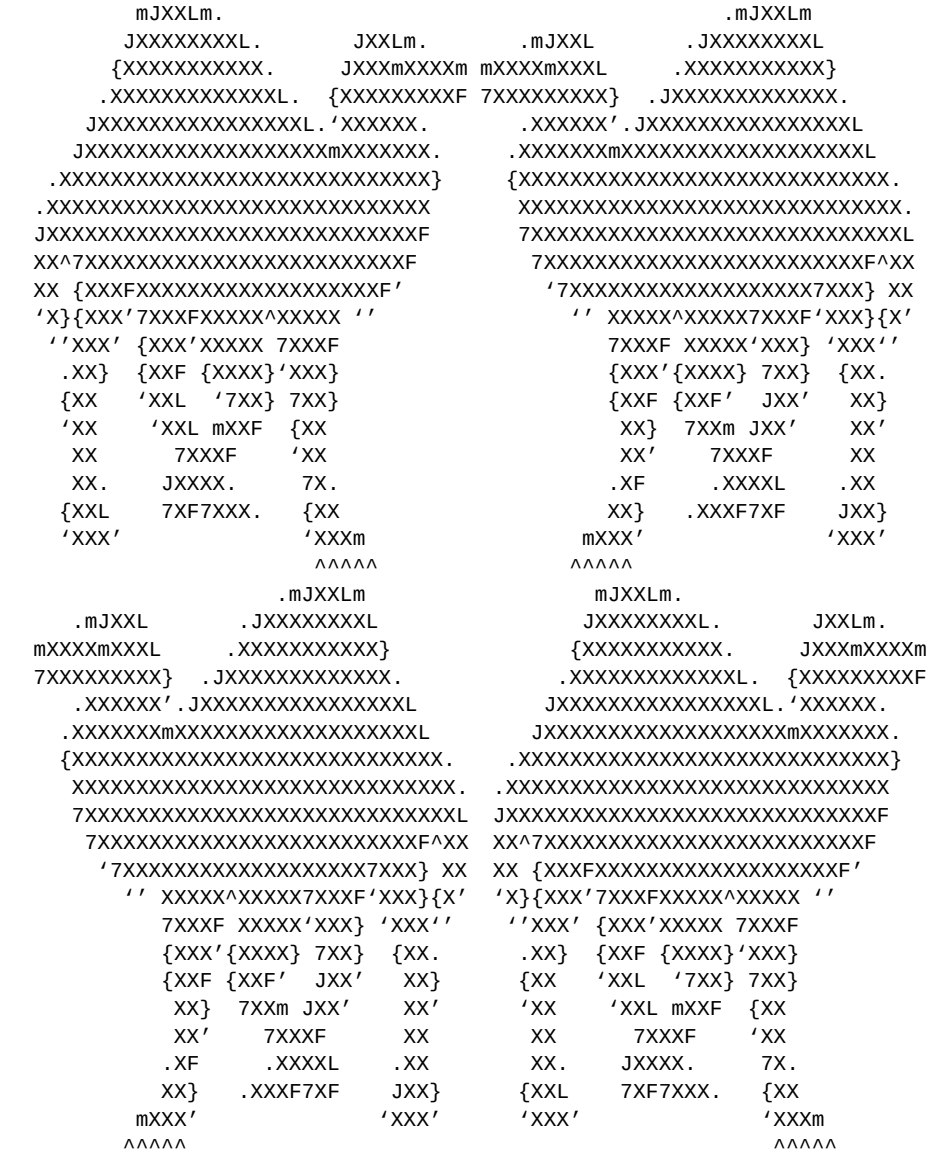
CTRL-G from the main menu, or just read the menu bar.

Appendix G. ASCII Pronunciation Guide

Table G-1. ASCII Pronunciation Guide

Character	Pronunciation
#	hash, pound, sharp, number
!	bang, exclamation
*	star, asterisk
$	dollar
@	at
%	percent, percentage sign
&	ampersand
"	double-quote
'	single-quote, tick, forward tick
( )	open/close parentheses, round brackets, bananas
<	less than
>	greater than
-	dash, hyphen
.	dot
,	comma
/	slash, forward-slash
\	backslash, slosh
:	colon
;	semi-colon
=	equals
?	question-mark
^	caret (pron. carrot), hat
_	underscore
[ ]	open/close bracket, square bracket
{ }	open/close curly brackets, brace
|	pipe, vertical bar, bar
~	tilde, wiggle, squiggle
`	backtick

Colophon



The use of a camel image in association with Perl is a trademark of O’Reilly & Associates, Inc. Used with permission.

The `camel code` that makes up the cover art was written by Stephen B. Jenkins (aka Erudil). When executed, it generates the images of four smaller camels as shown above. A discussion of the camel code in its native habitat can be found on PerlMonks (http://www.perlmonks.org/index.pl?node=camel+code). More information about Stephen B. Jenkins and his work can be found on his website (http://www.Erudil.com).