

Introduction to Perl

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

    $_='$;="per
1";map{map {s^\s+}}
;$_{$_}++unless(/[^a-
z]/)}split(/
[\s,]+/)i
f(/alpha.
*$/i../w
ait/)}'$_;
doc\040$_;
toc';;@[=k
eys% ;$_;      =20;$_:=15;for(0..($_*$_-1
))){$_{$_}="
";}until($%++>3*$_||@>2*$_-3){@_ =split(/,splice(@[,rand(
@[,1)]);if(3>@_){next;}$_=int(rand($_));$^=int(rand($_));
$_=$-+$_*$_;my$Erudil=0;{if($Erudil++>2*$_){next;}$a=(-1,
0,1)[rand(3)];$b=(-1,0,1)[rand(3)];unless(($a||$b)&&$~
+$_*@_<=$&&$~+$_*@_>=0&&$^+$_*@_<=$&&$^+$_*@_>=0){re
do;;}my$llama=0;for(0..$_){unless($_{$_+$_*$_+$_*
$_*$_leq$_{$_}||$_{$_+$_*$_+$_*$_*$_leq$_}){$llam
a++;last;}}if($llama){redo;push@,join(" " @_);f
or(0..$_){$_{$_+$_*$_+$_*$_*$_*$_leq$_{$_}||$_}
=sort@;unshift@_,"Find:","-x5;for$a(0.
.$:-1){for$b(0.
.$:-1){$_~("a.."z")
[rand(26)];$_
=$";s;$_;$_;
;print;}$_=$
hifft@_|$_;print$
",$_,"
$_,$ /;$_
=shi      ft@_|$_
";pr      int  "$x
$_;       "$x  $_;
$_;       "$"  $_
.$;/;;    };   ;;;
s[\s+]    $$g; eval;
```

Kirrily Robert
Paul Fenwick
Jacinta Richardson

Introduction to Perl

by KIRRILY ROBERT, PAUL FENWICK, and JACINTA RICHARDSON

Copyright © 1999-2000 Netizen Pty Ltd

Copyright © 2000 KIRRILY ROBERT

Copyright © 2001 Obsidian Consulting Group Pty Ltd

Copyright © 2001-2005 Paul Fenwick (pjf@perltraining.com.au)

Copyright © 2001-2005 Jacinta Richardson (jarich@perltraining.com.au)

Copyright © 2001-2005 Perl Training Australia

Open Publications License 1.0

Cover artwork Copyright (c) 2000 by Stephen B. Jenkins. Used with permission.

The use of a llama image with the topic of Perl is a trademark of O'Reilly & Associates, Inc. Used with permission.

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

Distribution of this work or derivative of this work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

This document is a revised and edited copy of the 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 revision 4.20 of the Perl Training Australia's "Introduction to Perl" training manual.

Table of Contents

1. About Perl Training Australia	1
Training	1
Consulting	1
Contact us	1
2. Introduction.....	3
Credits	3
Course outline	3
Day 1	3
Day 2	3
Assumed knowledge	3
Module objectives	3
Platform and version details	4
The course notes.....	4
Other materials	5
3. What is Perl	7
In this chapter.....	7
Perl's name and history	7
Typical uses of Perl	7
Text processing	7
System administration tasks	7
CGI and web programming	7
Database interaction	8
Other Internet programming.....	8
Less typical uses of Perl	8
What is Perl like?	8
The Perl Philosophy	9
There's more than one way to do it	9
A correct Perl program.....	9
Three virtues of a programmer	9
Laziness.....	9
Impatience.....	9
Hubris.....	9
Three more virtues.....	10
Share and enjoy!	10
Parts of Perl	10
The Perl interpreter	10
Manuals/Documentation.....	10
Perl Modules.....	11
Chapter summary	11
4. A brief guide to perldoc.....	13
Using perldoc	13
Exercise	13
Language features and tutorials	13
Looking up functions	13
Looking up modules.....	14

5. Creating and running a Perl program.....	15
In this chapter.....	15
Logging into your account	15
Our first Perl program	15
Running a Perl program from the command line.....	15
Executing code.....	16
The "shebang" line for Unix.....	16
The "shebang" line for non-Unixes	17
Comments	17
Command line options	17
Chapter summary	17
6. Perl variables.....	19
In this chapter.....	19
What is a variable?.....	19
Variable names	19
Variable scoping and the strict pragma	19
Arguments in favour of strictness.....	20
Arguments against strictness	20
Using the strict pragma (predeclaring variables).....	20
Exercise.....	21
Using the diagnostics pragma.....	21
Exercise.....	21
Starting your Perl program	21
Scalars	22
Double and single quotes	23
Exercise	23
Special characters	23
Advanced Variable Interpolation.....	24
Exercises.....	24
Arrays.....	24
Initialising an array.....	25
Reading and changing array values	25
Array slices	25
Array interpolation	26
Counting backwards	26
Finding out the size of an array	26
Using qw// to populate arrays.....	27
Printing out the values in an array	27
A quick look at context.....	27
What's the difference between a list and an array?	28
Exercises.....	29
Advanced exercises	29
Hashes	29
Initialising a hash.....	29
Reading hash values	30
Adding new hash elements	30
Changing hash values	31
Deleting hash values.....	31
Finding out the size of a hash.....	31
Other things about hashes.....	31
Exercises.....	31

Special variables.....	32
The special variable \$_.....	32
Exercises	33
@ARGV - a special array.....	33
Exercise.....	33
%ENV - a special hash.....	33
Exercises	33
Chapter summary	34
7. Operators and functions.....	35
In this chapter.....	35
What are operators and functions?.....	35
Operators	35
Arithmetic operators.....	35
String operators	36
Exercises.....	37
File test operators	37
Other operators	37
Functions	38
Types of arguments.....	39
Return values	39
More about context	40
Some easy functions.....	40
String manipulation	41
Finding the length of a string	41
Case conversion	41
chop() and chomp().....	41
String substitutions with substr()	42
Exercises	42
Numeric functions	43
Type conversions	43
Manipulating lists and arrays.....	43
Stacks and queues	44
Ordering lists.....	44
Converting lists to strings, and vice versa.....	44
Exercises.....	45
Hash processing.....	45
Reading and writing files.....	45
Time.....	46
Exercises.....	46
Chapter summary	46
8. Conditional constructs.....	47
In this chapter.....	47
What is a conditional statement?	47
What is truth?	47
The if conditional construct	47
So what is a BLOCK?	48
Exercises.....	49
Scope	49
Comparison operators	50
Exercises	51
Existence and definitiveness.....	51

Exercise	52
Boolean logic operators	53
Using boolean logic operators as short circuit operators	54
Boolean assignment	55
Loop conditional constructs	55
while loops.....	55
do while loops.....	56
for and foreach.....	56
Exercises.....	57
Practical uses of <code>while</code> loops: taking input from STDIN	57
Exercises.....	58
Named blocks.....	59
Breaking out or restarting loops.....	59
Practical exercise.....	60
Chapter summary	60
9. Subroutines.....	63
In this chapter.....	63
Introducing subroutines.....	63
What is a subroutine?	63
Why use subroutines?.....	63
Using subroutines in Perl.....	63
Calling a subroutine	64
Passing arguments to a subroutine	64
Passing in scalars	65
Passing in arrays and hashes.....	65
Returning values from a subroutine	66
Exercises	67
Chapter summary	68
10. Regular expressions	69
In this chapter.....	69
What are regular expressions?.....	69
Regular expression operators and functions.....	69
m/PATTERN/ - the match operator	69
s/PATTERN/REPLACEMENT/ - the substitution operator.....	70
Exercises	70
Binding operators	71
Easy Modifiers.....	71
Meta characters	71
Some easy meta characters.....	71
Quantifiers	73
Exercises.....	73
Grouping techniques	74
Character classes	74
Exercises	75
Alternation.....	75
The concept of atoms.....	76
Exercises	76
Chapter summary	77

11. Practical exercises	79
12. Conclusion	81
What you've learnt	81
Where to now?	81
Further reading	81
Books	82
Online	82
A. Advanced Perl Variables	83
In this chapter	83
Quoting with <code>qq()</code> and <code>q()</code>	83
Exercises	84
Scalars in assignment	84
Arrays in assignment	85
Hash slices	86
Exercise	87
Hashes in assignment	87
Chapter summary	88
B. Named parameter passing and default arguments	89
In this chapter	89
Named parameter passing	89
Default arguments	90
Exercises	90
Subroutine declaration and prototypes	90
Chapter summary	91
C. Unix cheat sheet	93
D. Editor cheat sheet	95
vi (or vim)	95
Running	95
Using	95
Exiting	95
Gotchas	95
Help	96
nano (pico clone)	96
Running	96
Using	96
Exiting	96
Gotchas	96
Help	96
E. ASCII Pronunciation Guide	97
Colophon	99

List of Tables

1-1. Perl Training Australia's contact details.....	1
4-1. Getting around in perldoc.....	13
6-1. Variable punctuation.....	19
7-1. Arithmetic operators.....	35
7-2. String operators	36
7-3. File test operators	37
7-4. Context-sensitive functions	40
8-1. Numerical comparison operators.....	50
8-2. String comparison operators.....	50
8-3. Boolean logic operators.....	53
10-1. Binding operators	71
10-2. Regexp modifiers.....	71
10-3. Regular expression meta characters	72
10-4. Regular expression quantifiers	73
C-1. Simple Unix commands.....	93
D-1. Layout of editor cheat sheets	95
E-1. ASCII Pronunciation Guide.....	97

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 *Introduction to Perl* training module. This is a two-day training module in which you will learn how to program in the Perl programming language.

Credits

This course is based upon the Introduction to Perl training module 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

Assumed knowledge

To gain the most from this course, you should:

- Have programmed in least one other language and
 - Understand variables, including data types and arrays
 - Understand conditional and looping constructs
 - Understand the use of subroutines and/or functions

Module objectives

- Understand the history and philosophy behind the Perl programming language
- Know where to find additional information about Perl
- Write simple Perl scripts and run them from the Unix command line
- Use Perl's command line options to enable warnings
- Understand Perl's three main data types and how to use them
- Use Perl's `strict` pragma to enforce lexical scoping and better coding
- Understand Perl's most common operators and functions and how to use them
- Understand and use Perl's conditional and looping constructs
- Understand and use subroutines in Perl
- Understand and use simple regular expressions for matching and substitution

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.

All of Perl Training Australia's Perl training courses use Perl 5, the most recent major release of the Perl language. Perl 5 differs significantly from previous versions of Perl, so you will need a Perl 5 interpreter to use what you have learnt. However, older Perl programs should work fine under Perl 5.

At the time of writing, the most recent stable release of Perl is version 5.8.6, however older versions of Perl 5 are still common. Your instructor will inform you of any features which may not exist in older 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". This book can be found at most major bookstores. Alternately, if you're planning on attending a Perl Training Australia course in the future, ask about our discount rates for this and other suggested text books.

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. 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 E 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 to 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 interpreter 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.



book.

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

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 (see pages 103-104 (pages 90-91, 2nd Ed) of your Camel book or **perldoc perlop**) which creates a temporary list of numbers between the two you specify - in this case 0 and 1, but it could have been 1 and 100 if we'd had an array big enough to use it on. You'll run into this operator again and again, so remember it.

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 "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 $showmany = @array;           # scalar context
my @newarray = @array;          # list context
```

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.



book.

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

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".



book.

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

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

We can print out the arguments given to a program with a script like the following:

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

print "@ARGV\n";
```

This listing is in the `exercises/printargs.pl` script in your directory. Run the script now, passing in the names of your friends on the command line.

%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
+	<code>\$a + \$b</code>	Addition
-	<code>\$a - \$b</code>	Subtraction

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



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
.	<code>\$a . \$b</code>	Concatenation (puts <code>\$a</code> and <code>\$b</code> together as one string)
x	<code>\$a x \$b</code>	Repeat (repeat <code>\$a</code> <code>\$b</code> times --- eg <code>"foo" x 3</code> gives us <code>"foofoofoo"</code>)



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
<code>-e</code>	<code>-e \$a</code>	Exists - does the file exist?
<code>-r</code>	<code>-r \$a</code>	Readable - is the file readable?
<code>-w</code>	<code>-w \$a</code>	Writable - is the file writable?
<code>-d</code>	<code>-d \$a</code>	Directory - is it a directory?
<code>-f</code>	<code>-f \$a</code>	File - is it a normal file?
<code>-T</code>	<code>-T \$a</code>	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
<code>reverse()</code>	Reverses characters in a string.	Reverses the order of the elements in an array.
<code>each()</code>	Returns the next key in a hash.	Returns a two-element list consisting of the next key and value pair in a hash.
<code>gmtime()</code> and <code>localtime()</code>	Returns the time as a string in common format.	Returns a list of second, minute, hour, day, etc.
<code>keys()</code>	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.
<code>readdir()</code>	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 `chomp()` on 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 `pushing` items onto the end of an array and `shifting` them off the front.

A stack can be created by `pushing` items on the end of an array and `poping` 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.

Exercises

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

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

- 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`)
- 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 in our Intermediate Perl 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.
- Chapter 29 (Chapter 3, 2nd Ed) of your Camel book, **`perldoc perlop`**, **`perldoc perlfunc`**, and **`perldoc -f functionname`** can be used to find out detailed information about operators and functions.
- 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 goes 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          # make sure we have apples
if( not $I_have_apples ) {          unless( $I_have_apples ) {
    go_buy_apples();                go_buy_apples();
}                                     }

# now that we have apples...        # now that we have apples...
make_apple_pie();                   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 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";
    $showmany = 32;
    print "I'd like to buy $showmany ${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";
{
    my $a = "bar";          # start a new block
                           # 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
<code>==</code>	<code>\$a == \$b</code>	Equality (same as in C and other C-like languages)
<code>!=</code>	<code>\$a != \$b</code>	Inequality (again, C-like)
<code><</code>	<code>\$a < \$b</code>	Less than
<code>></code>	<code>\$a > \$b</code>	Greater than
<code><=</code>	<code>\$a <= \$b</code>	Less than or equal to
<code>>=</code>	<code>\$a >= \$b</code>	Greater than or equal to
<code><=></code>	<code>\$a <=> \$b</code>	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
<code>eq</code>	<code>\$a eq \$b</code>	Equality
<code>ne</code>	<code>\$a ne \$b</code>	Inequality

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

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:

- If the number is less than 3, print "Too small"
- If the number is greater than 7, print "Too big"
- 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
<code>and</code>	<code>&&</code>	<code>\$a && \$b</code> <code>\$a and \$b</code>	True if both <code>\$a</code> and <code>\$b</code> are true; acts on <code>\$a</code> then if <code>\$a</code> is true, goes on to act on <code>\$b</code> .
<code>or</code>	<code> </code>	<code>\$a \$b</code> <code>\$a or \$b</code>	True if either of <code>\$a</code> and <code>\$b</code> are true; acts on <code>\$a</code> then if <code>\$a</code> is false, goes on to act on <code>\$b</code> .
<code>not</code>	<code>!</code>	<code>! \$a</code> <code>not \$a</code>	True if <code>\$a</code> is false. False if <code>\$a</code> 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 a 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 than 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 incredibly 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 `shifting` 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 and 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 course.



To learn more about references attend our *Intermediate Perl* course or 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
<code> =~ </code>	True if the pattern matches
<code> !~ </code>	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
<code>/i</code>	Make match/substitute match case insensitive
<code>/g</code>	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...
<code>^</code>	Start of string
<code>\$</code>	End of string
<code>.</code>	Any single character except <code>\n</code>
<code>\n</code>	Newline
<code>\t</code>	Matches a tab
<code>\s</code>	Any whitespace character, such as space, tab, or newline
<code>\S</code>	Any non-whitespace character
<code>\d</code>	Any digit (0 to 9)
<code>\D</code>	Any non-digit
<code>\w</code>	Any "word" character - alphanumeric plus underscore (<code>_</code>)
<code>\W</code>	Any non-word character
<code>\b</code>	A word break - the zero-length point between a word character (as defined above) and a non-word character.
<code>\B</code>	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 in the Intermediate Perl 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. 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 12. Conclusion

What you've learnt

Now you've completed Perl Training Australia's Introduction to Perl module, you should be confident in your knowledge of the following fields:

- What is Perl? Perl's features, Perl's main uses, where to find information about Perl online.
- Creating Perl scripts and running them from the Unix command line, including the use of the `-w` flag to enable warnings.
- Perl's three main variable types: scalars, arrays and hashes.
- The `strict` pragma, lexical scoping, and their benefits.
- Perl's concept of truth; existence and definitiveness of variables.
- Conditional and looping constructs: `if`, `unless`, `while`, `foreach` and others.
- Regular expressions: the matching and substitution operators; meta characters; quantifiers; alternation and grouping.

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:
 - Intermediate Perl
 - CGI Programming 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

- Randal L. Schwartz and Tom Phoenix, *Learning Perl* (3rd Ed), O'Reilly and Associates, 2001. ISBN 0-596-00132-0
- 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.

Online

- The Perl homepage (<http://www.perl.com/>)
- The Perl Journal (<http://www.tpj.com/>)
- Perlmonth (<http://www.perlmonth.com/>) (online journal)
- Perl Mongers Perl user groups (<http://www.pm.org/>)
- PerlMonks online community (<http://www.perlmonks.org/>)
- comp.lang.perl.announce newsgroup
- comp.lang.perl.moderated newsgroup
- comp.lang.perl.misc newsgroup
- Comprehensive Perl Archive Network (<http://www.cpan.org>)

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);
```

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 the 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 way 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. Unix cheat sheet

A brief run-down for those whose Unix skills are rusty:

Table C-1. Simple Unix commands

Action	Command
Change to home directory	cd
Change to <i>directory</i>	cd <i>directory</i>
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 <i>filename</i>
Printing a long file a screenful at a time	more <i>filename</i> or less <i>filename</i>
Getting help for <i>command</i>	man <i>command</i>

Appendix D. Editor cheat sheet

This summary is laid out as follows:

Table D-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 E. ASCII Pronunciation Guide

Table E-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, slos
:	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

```
m m u q y j t i m e w w m y n j v i k n   Find:
-----
k s c f h m p o h c v e n r d q t r o s   alarm
f i f m s o c k e t n i d j j f g u t t   binmode
n n i b p h c w f c s k e o v s e v w s   cate
o p r y c o n t i n u e y t m i i n p j   chop
a k s m s e e y j o l v u a t n y u s c   continue
m n t r t m t k i l l t n n m j i l n i   exec
w i e a s i a q k a d r u a n s a b d h   exists
s l x l i t c b o i g h t g n e l k v a   fileno
f i e a x m p t w t s k n o o y x p y o   getgrnam
o p c o e g o b e b j v j n u j h i p q   getpwnam
n p s d u g d g e o w k e h o v t f j s   gmtime
t i m p o r t e c v p l g d t v u r f x   goto
c x v n n e x t o k i g e t p w n a m d   import
k p d o q b b z p f i r y m a p s e n d   int
                                         kill
                                         length
                                         link
                                         map
                                         next
                                         oct
                                         redo
                                         send
                                         sin
                                         socket
                                         sort
                                         time
                                         ucfirst
                                         untie
```

The `find-a-func` (also known as the *llama code*) that makes up the cover art was written by Stephen B. Jenkins (aka Erudil). When executed, it generates a (random) "find-a-func" puzzle, as demonstrated above. A discussion of the `find-a-func` code in its native habitat can be found on PerlMonks (http://perlmonks.org/index.pl?node_id=108730). More information about Stephen B. Jenkins and his work can be found on his website (<http://www.Erudil.com/>).

