Programming Perl

```
#!/usr/bin/perl −w
use strict;
                                           $_=’ev
                                       al("seek\040D
           ATA,0,                  0;");foreach(1..2)
       {<DATA>;}my               @camel1hump;my$camel;
  my$Camel  ;while(             <DATA>){$_=sprintf("%−6
9s",$_);my@dromedary           1=split(//);if(defined($
_=<DATA>)){@camel1hum        p=split(//);}while(@dromeda
 ry1){my$camel1hump=0      ;my$CAMEL=3;if(defined($_=shif
        t(@dromedary1    ))&&/\S/){$camel1hump+=1<<$CAMEL;}
       $CAMEL−−;if(d   efined($_=shift(@dromedary1))&&/\S/){
      $camel1hump+=1  <<$CAMEL;}$CAMEL−−;if(defined($_=shift(
     @camel1hump))&&/\S/){$camel1hump+=1<<$CAMEL;}$CAMEL−−;if(
     defined($_=shift(@camel1hump))&&/\S/){$camel1hump+=1<<$CAME
     L;;}$camel.=(split(//,"\040..m‘{/J\047\134}L^7FX"))[$camel1h
      ump];}$camel.="\n";}@camel1hump=split(/\n/,$camel);foreach(@
      camel1hump){chomp;$Camel=$_;tr/LJF7\173\175‘\047/\061\062\063
      45678/;tr/12345678/JL7F\175\173\047‘/;$_=reverse;print"$_\040
       $Camel\n";}foreach(@camel1hump){chomp;$Camel=$_;y/LJF7\173\17
        5‘\047/12345678/;tr/12345678/JL7F\175\173\047‘/;$_=reverse;p
         rint"\040$_$Camel\n";}#japh−Erudil’;;s;\s*;;g;;eval;   eval
           ("seek\040DATA,0,0;");undef$/;$_=<DATA>;s$\s*$$g;(   );;s
             ;^.*_;;;map{eval"print\"$_\"";}/.{4}/g; __DATA__   \124
               \1   50\145\040\165\163\145\040\157\1 46\040\1  41\0
                    40\143\141  \155\145\1 54\040\1   51\155\  141
                    \147\145\0  40\151\156 \040\141    \163\16 3\
                     157\143\   151\141\16  4\151\1     57\156
                     \040\167  \151\164\1   50\040\      120\1
                     45\162\   154\040\15    1\163\      040\14
                     1\040\1   64\162\1      41\144       \145\
                     155\14    1\162\       153\04        0\157
                      \146\     040\11     7\047\         122\1
                      45\15      1\154\1  54\171          \040
                      \046\         012\101\16            3\16
                      3\15           7\143\15             1\14
                      1\16            4\145\163           \054
                     \040            \111\156\14         3\056
                    \040\         125\163\145\14         4\040\
                    167\1        51\164\1  50\0         40\160\
                  145\162                              \155\151
                \163\163                                \151\1
              57\156\056
```

Kirrily Robert
Paul Fenwick
Jacinta Richardson

Programming Perl

by Kirrily Robert, Paul Fenwick, and Jacinta Richardson

Open Publications License 1.0

This document is a revised and edited copy of the Introduction to Perl and Intermediate Perl training notes originally created by Kirrily Robert and Netizen Pty Ltd. These revisions were made by Paul Fenwick and Jacinta Richardson.

Copies of the original training manuals can be found at http://sourceforge.net/projects/spork

This training manual is maintained by Perl Training Australia, and can be found at http://www.perltraining.com.au/notes.html

This is version 1.36 of Perl Training Australia’s "Programming 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
  - Day 3 ... 3
  - Day 4 ... 3
  - Day 5 ... 4
- Assumed knowledge ... 4
- Platform and version details ... 4
- The course notes ... 4
- Other materials ... 5

3. What is Perl ... 7

- In this chapter... ... 7
- License ... 7
- Perl's name and history ... 7
- Typical uses of Perl ... 7
  - Text processing ... 7
  - System administration tasks ... 7
  - CGI and web programming ... 8
  - 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 ... 10
  - Three more virtues ... 10
  - Share and enjoy! ... 10
- Parts of Perl ... 10
  - The Perl interpreter ... 10
  - Manuals/Documentation ... 11
  - Perl Modules ... 11
- Chapter summary ... 11

4. A brief guide to perldoc ... 13

- Using perldoc ... 13
  - Other ways to access perldoc ... 13
  - Exercise ... 13
- Language features and tutorials ... 13
- Looking up functions ... 14

Searching the FAQs 14
Looking up modules 14
5. Creating and running a Perl program 17
In this chapter... 17
Logging into your account 17
Our first Perl program 17
Running a Perl program from the command line 17
Executing code 18
The "shebang" line for Unix 18
The "shebang" line for non-Unixes 19
Command line options and warnings 19
Lexical warnings 19
Comments 20
Block comments 20
__END__ 20
Chapter summary 20
6. Perl variables 23
In this chapter... 23
What is a variable? 23
Variable names 23
Variable scoping and the strict pragma 23
Arguments in favour of strictness 24
Arguments against strictness 24
Using the strict pragma (predeclaring variables) 24
Exercise 25
Using the diagnostics pragma 25
splain 25
Exercise 25
Starting your Perl program 26
Scalars 26
Double and single quotes 27
Exercise 28
Special characters 28
Advanced variable interpolation 28
Exercises 29
say (Perl 5.10.0+) 29
Arrays 30
Initialising an array 30
Reading and changing array values 30
Array slices 30
Array interpolation 31
Counting backwards 31
Finding out the size of an array 31
Using qw// to populate arrays 32
Printing out the values in an array 33
A quick look at context 33
What’s the difference between a list and an array? 34
Exercises 34
Advanced exercises 34
Hashes 34
Initialising a hash 35

Reading hash values 35
Adding new hash elements 36
Changing hash values 36
Deleting hash values 36
Finding out the size of a hash 36
Other things about hashes 36
Exercises 37
Special variables 38
The special variable $_ 38
@ARGV - a special array 38
%ENV - a special hash 39
Exercises 39
Programming practices 40
Perltidy 40
Perl Critic 41
Chapter summary 41
7. Operators and functions 43
In this chapter... 43
What are operators and functions? 43
Operators 43
Arithmetic operators 43
String operators 44
Exercises 45
Advanced exercise 45
Other operators 45
Functions 45
Types of arguments 46
Return values 47
More about context 47
Some easy functions 48
String manipulation 48
Finding the length of a string 48
Case conversion 48
chop() and chomp() 49
String substitutions with substr() 49
Exercises 50
Numeric functions 50
Type conversions 51
Manipulating lists and arrays 51
push, pop, shift and unshift 51
Ordering lists 52
Converting strings to lists, and vice versa (split and join) 52
Exercises 53
Hash processing 53
Reading and writing files 53
Time 53
Exercises 54
Chapter summary 54

8. Conditional constructs......55
- In this chapter... 55
- What is a conditional statement? 55
- What is truth? 55
- The if conditional construct 55
  - So what is a `BLOCK`? 56
  - Scope 57
- Comparison operators 58
- Exercises 59
- Existence and definitiveness......59
- Exercise 60
- Boolean logic operators 61
- Logic operators and short circuiting 62
- Boolean assignment 63
- Loop conditional constructs 63
  - while loops......64
  - for and foreach......64
  - Exercises......66
- Practical uses of `while` loops: taking input from STDIN 66
  - Exercises......67
- Named blocks......67
- Breaking out or restarting loops......67
- Practical exercise......69
- Smart-match......69
- given and when......70
- Chapter summary 71

9. Subroutines......73
- In this chapter... 73
- Introducing subroutines......73
  - What is a subroutine? 73
  - Why use subroutines?......73
  - Using subroutines in Perl......73
- Calling a subroutine 74
- Passing arguments to a subroutine 75
  - Passing in scalars......75
  - Passing in arrays and hashes......76
- Returning values from a subroutine 77
- Exercises 78
- Chapter summary 78

10. Named parameter passing and default arguments......81
- In this chapter... 81
- Named parameter passing 81
- Default arguments 82
- Subroutine declaration and prototypes......82
- Chapter summary 83

11. Regular expressions 85
In this chapter... 85
What are regular expressions? 85
Regular expression operators and functions 85
m/PATTERN/ - the match operator 85
s/PATTERN/REPLACEMENT/ - the substitution operator 86
Exercises 86
Binding operators 86
Easy modifiers 87
Meta characters 87
Some easy meta characters 87
Quantifiers 89
Exercises 90
Grouping techniques 90
Character classes 90
Exercises 91
Alternation 91
The concept of atoms 92
Exercises 93
Chapter summary 93
12. Introduction to Modules 95
In this chapter... 95
What is a module? 95
Module uses 95
use statements 95
Lexically scoped modules 96
Passing in directives 96
The double-colon 97
Exercise 97
The Comprehensive Perl Archive Network (CPAN) 97
Exercise 97
CPAN features 97
Using CPAN modules 98
Installing CPAN modules to the system directories 98
Installing manually from a tar ball 98
Installing manually with PPM (ActivePerl on Windows) 98
Installing manually with cpan (Unix and Windows) 99
Installing manually with cpanminus 99
Exercise 99
Finding installed modules 99
How does Perl find the modules? 100
Adding paths to @INC 100
Relative paths 101
Working with a local install directory 102
Manually installing modules locally 102
Installing modules locally with `cpan` 102
local::lib and cpanm 103
cpanm 103
Microsoft Windows 103
PAR 104
Finding quality CPAN modules 104

Chapter summary 104
13. Using Perl objects 105
In this chapter... 105
Objects in brief 105
Loading a class 105
Instantiating an object 106
Calling methods on an object 106
Destroying an object 107
Chapter summary 107
14. Testing your module 109
Testing Strategies 109
Black box testing 109
White box testing 110
Combining these ideas 110
Our project 110
Exercise 110
Running our test suite 111
Exercise 112
Writing our own tests with Test::More 112
Having a plan 113
SKIP and TODO blocks 114
Test data 114
Exercise 115
Coverage testing and Devel::Cover 116
Exercise 117
Good Tests 117
Chapter summary 118
15. External Files and Packages 119
In this chapter... 119
Splitting code between files 119
Require 119
Use strict and warnings 120
Example 120
Exercises 121
Introduction to packages 121
The scoping operator 122
Package variables and our 123
Exercises 124
Chapter summary 124
16. Writing modules 127
In this chapter... 127
The rules for writing a module 127
Module naming 127
Use versus require 128
Warnings and strict 128
Exercise 128
Making it into a module 129
Moving your module into a subdirectory 129
Things to remember... 129
Exporting and importing subroutines 129

@ISA 130
use base 130
An example 130
Exporting by default 130
An example 131
Importing symbols 132
Exercises 132
Exporting tags 132
Importing symbols through tags 132
Writing your own documentation 133
Exercise 135
Chapter summary 135
17. Building modules with Module::Starter 137
In this chapter... 137
Using module-starter 137
Configuration 137
Exercise 138
A skeleton tour 138
Local Installation 139
Exercise 139
Further information 140
Chapter Summary 140
18. References and complex data structures 141
In this chapter... 141
Assumed knowledge 141
Introduction to references 141
Uses for references 141
Creating complex data structures 141
Passing arrays and hashes to subroutines and functions 141
Object oriented Perl 142
Creating and dereferencing references 142
Exercises 143
Assigning through references 144
Passing multiple arrays/hashes as arguments 144
Anonymous data structures 145
Exercise 146
Complex data structures 146
Exercises 147
Dereferencing expressions 148
Copying complex data structures 148
Data::Dumper 149
Exercises 151
Chapter summary 151
19. Advanced regular expressions 153
In this chapter... 153
Assumed knowledge 153
Capturing matched strings to scalars 153
Extended regular expressions 154
Exercise 155
Advanced exercise 155

Greediness ... 156
Exercise ... 156
More meta characters ... 156
Working with multi-line strings ... 157
Exercise ... 159
Regexp modifiers for multi-line data ... 159
Regular expression special variables ... 160
PREMATCH, MATCH and POSTMATCH ... 160
Efficiency concerns ... 161
Perl 5.10+ alternative ... 161
Capture variables ... 161
Back-references ... 162
Exercises ... 162
Advanced exercises ... 163
Chapter summary ... 163
20. File I/O ... 165
In this chapter... ... 165
Angle brackets ... 165
The line input operator ... 165
Opening a file for reading, writing or appending ... 165
Opening and reading from a file ... 166
Opening files for writing and appending ... 166
Closing your file ... 167
Failure ... 167
Using autodie to handle failure ... 168
Exercises ... 169
Package filehandles (legacy) ... 169
Scalar filehandles before Perl 5.6.0 ... 170
Two argument open ... 171
Changing file contents ... 171
Secure temporary files ... 172
Looping over file contents ... 173
Exercises ... 173
Opening files for simultaneous read/write ... 173
The small print ... 174
Buffering ... 174
Opening pipes ... 175
Security issues ... 176
Exercises ... 177
File locking ... 178
Handling binary data ... 179
Chapter summary ... 180
21. System interaction ... 181
In this chapter... ... 181
Platform dependency issues ... 181
system() ... 181
Error handling ... 181
Security issues ... 182
Multi-argument system ... 182
exec ... 183
IPC::System::Simple and autodie ... 183

Capturing the exception 183
With autodie 184
autodie qw(:all) 184
Allowable return values 185
IPC::System::Simple and the shell 185
systemx 185
Exercises 186
*nix exercises 186
MS Windows exercise 186
Using backticks 186
Backticks vs system() 186
Security issues 187
Capturing with IPC::System::Simple 187
Exercises 188
*nix exercises 188
MS Windows exercises 188
Using tainted data (Perl’s security model) 188
Insecure code 189
Which program? 189
Explicitly setting $ENV{PATH} 189
Turning on taint 190
Cleaning tainted data 190
Exercise 191
Chapter summary 191
22. Practical exercises 193
About these exercises 193
Palindromes 193
Hangman 193
23. Conclusion 195
Where to now? 195
Further reading 195
Books 195
Online 195
A. Advanced Perl variables 197
In this chapter... 197
Quoting with `qq()` and `q()` 197
Exercises 198
Scalars in assignment 198
Arrays in assignment 199
Hash slices 200
Searching for items in a large list 200
Unique values 201
Exercise 201
Hashes in assignment 201
Chapter summary 202

B. Complex data structures 205
- Arrays of arrays 205
  - Creating and accessing a two-dimensional array 205
  - Adding to your two-dimensional array 205
  - Printing out your two-dimensional array 206
- Hashes of arrays 206
  - Creating and accessing a hash of arrays 206
  - Adding to your hash of arrays 206
  - Printing out your hash of arrays 207
- Arrays of hashes 207
  - Creating and accessing an array of hashes 207
  - Adding to your array of hashes 207
  - Printing out your array of hashes 208
- Hashes of hashes 208
  - Creating and accessing a hash of hashes 208
  - Adding to your hash of hashes 209
  - Printing out your hash of hashes 209
- More complex structures 209

C. Directory interaction 211
- In this chapter... 211
- Reading a direction with glob() 211
  - Exercises 211
- Finding information about files 211
  - Multiple file tests 213
  - Exercises 213
- Changing the working directory 213
- Recursing down directories 214
  - File::Find::Rule 215
  - Exercises 215
- opendir and readdir 216
  - Package directory handles 216
  - Exercises 217
- glob and readdir 217
- Chapter summary 217

D. More functions 219
- The grep() function 219
  - Exercises 219
- The map() function 219
  - Exercises 220
- List manipulation with splice() 220

E. Unix cheat sheet 221

F. Editor cheat sheet 223
- vi (or vim) 223
  - Running 223
  - Using 223
  - Exiting 223
  - Gotchas 223
  - Help 224
- nano (pico clone) 224
  - Running 224

Using..........224
Exiting..........224
Gotchas..........224
Help..........224

G. ASCII Pronunciation Guide..........225

Colophon..........227

List of Tables

1-1. Perl Training Australia’s contact details 1
4-1. Getting around in perldoc 13
6-1. Variable punctuation 23
7-1. Arithmetic operators 43
7-2. String operators 44
7-3. Context-sensitive functions 48
8-1. Numerical comparison operators 58
8-2. String comparison operators 58
8-3. Boolean logic operators 61
11-1. Binding operators 87
11-2. Regexp modifiers 87
11-3. Regular expression meta characters 88
11-4. Regular expression quantifiers 89
19-1. More meta characters 157
19-2. Effects of single and multi-line options 160
C-1. File test operators 212
C-2. Differences between glob and readdir 217
E-1. Simple Unix commands 221
F-1. Layout of editor cheat sheets 223
G-1. ASCII Pronunciation Guide 225

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.

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:	+61 3 9354 6001
Fax:	+61 3 9354 2681
Email:	contact@perltraining.com.au
Webpage:	http://perltraining.com.au/
Address:	104 Elizabeth Street, Coburg VIC, 3058 AUSTRALIA

Chapter 2. Introduction

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

Credits

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

Course outline

Day 1

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

Day 2

- Conditional constructs
- Subroutines
- Named parameter passing
- Regular expressions

Day 3

- Introduction to modules and packages
- Writing packages and modules
- Building modules with Module::Starter
- Testing your module
- Using Perl objects

Day 4

- References and complex data structures
- Advanced regular expressions

Day 5

- File I/O
- System interaction and security
- Directory interaction
- Bonus material

Assumed knowledge

This training module assumes the following prior knowledge and skills:

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

Platform and version details

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

At the time of writing, the most recent stable release of Perl is 5.12.3, however older versions of Perl (particularly 5.8.x) are still common. Your instructor will inform you of any features which may not exist in earlier versions.

The course notes

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

The following typographical conventions are used in these notes:

System commands appear in **this typeface**

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

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

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

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

Notes and tips appear offset from the text like this.

Advanced sections 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.

Readme sections suggest pointers to more information which can be found in the *Programming Perl* book, system documentation such as manual pages and websites.

Caution sections contain details of unexpected behaviour or traps for the unwary.

Other materials

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

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

License

Perl is distributed under two licenses. These are the Artistic License and the GPL. You may choose which license you are using Perl under. For the text of these licenses read **perldoc perlartistic** and **perldoc perlgpl**.

Many of the modules that you can download from CPAN for Perl are also distributed under these same two licenses.

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 5.12.3, which is available from the Perl website (http://www.perl.com/). Perl is generally available through most package managers in *nix systems.

Windows users may find ActiveState's Active Perl (http://www.activestate.com/Products/activeperl/) and Strawberry Perl (http://strawberryperl.com/) to be good options.

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

This chapter discusses Perl’s extensive system documentation. 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.

Using perldoc

The most common way to access Perl’s documentation is via the **perldoc** system command that should have been installed when you installed Perl. We access this via a command prompt. **perldoc** will use your default pager, which on Windows is often **more** and on our Linux trainer server is **less**. The following instructions work in both pagers.

Table 4-1. Getting around in perldoc

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

Other ways to access perldoc

Depending on your operating system and how you installed Perl, you may also have other ways to access the perldoc pages. Additionally, you can access the perl documentation for several recent versions of Perl online (http://perldoc.perl.org/).

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.

In addition to the language, Perl comes with a set of utility programs. To find out more about these, read **perldoc perlutil**.

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**.

Searching the FAQs

Perl comes with several frequently asked questions (FAQ) files. These cover everything from general questions about Perl to using Perl for system interaction and networking. You can access these via perldoc: **perldoc perlfaq**, **perldoc perlfaq1**, **perldoc perlfaq2** and so on up to **perldoc perlfaq9**. Alternately you can search these with the `q` swtich: **perldoc -q <keyword>**. For example:

```
% perldoc -q round

Found in /usr/share/perl/5.8/pod/perlfaq4.pod
       Does Perl have a round() function?  What about ceil() and
       floor()?  Trig functions?

       Remember that int() merely truncates toward 0.  For rounding
       to a certain number of digits, sprintf() or printf() is
       usually the easiest route.
[...]
```

The `q` switch searches for your keyword in the text of the question, not of the answer. Where there are multiple matching questions, they will be displayed sequentially.

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.

Here are some useful **perldoc** tricks:

- To locate the install path of a particular module use **perldoc -l** ***`module_name`***.
- To view the source of a module use **perldoc -m** ***`module_name`***.
- To find all the modules installed on your system read **perldoc -q installed**.

Chapter 5. Creating and running a Perl program

In this chapter...

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

Logging into your account

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

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

Our first Perl program

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

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

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

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

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

Save it and exit.

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

Running a Perl program from the command line

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

```
% perl hello.pl
```

You should see this output:

```
Hello world!
```

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

Executing code

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

Well... it depends on the operating system.

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

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

The "shebang" line for Unix

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

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

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

We tell the operating system to use the Perl 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.

Command line options and warnings

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**.

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. The most commonly used option is `-w` to turn on warnings:

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

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

Lexical warnings

In Perl versions 5.6 and above you can use Perl's `warnings` pragma rather than the command line switch if you prefer. This also gives you the option to specify which warnings you wish to receive, and to upgrade those warnings to exceptions if necessary.

```
#!/usr/bin/perl
use warnings;
```

To learn more about this pragma read **perldoc perllexwarn** and **perldoc warnings**.

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 -w
# This is a hello world program
print "Hello world!\n";        # print the message
```

Block comments

To comment a block of text (or code) you can use Perl's Plain Old Documentation tags (POD). You can read more about POD in **perldoc perlpod**.

```
=begin comment

This content is commented out.
It may span many lines.

print "This statement won't be executed by Perl\n";

=end comment

=cut

print "Hello world!\n";        # print the message
```

__END__

You can signal to Perl the end of your program by using the special `__END__` tag. Anything below `__END__` will be ignored by Perl. This is particularly useful if you wish to include a large amount of documentation, or quickly comment out a large amount of code in one step.

```
print "Hello world!\n";        # print the message

__END__
All text and code from here downwards will be ignored by Perl.
print "This statement won't be executed by Perl\n";
```

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).
- Respect from other programmers. Because strict catches what is arguably the most commonly seen programming mistake (the typographical error), many senior developers won't take your code seriously, or help you debug your program, unless strict (and warnings) are turned on.

Arguments against strictness

- Takes a while to get used to for existing developers who have been using Perl without strict.
- 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
use strict;
use warnings;
```

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

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

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

my $number = 3;
```

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

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

Exercise

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

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

Using the diagnostics pragma

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

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

```
use diagnostics;
```

at the start of your code.

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

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

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

splain

An alternative to using the diagnostics pragma is to use **splain** which is a command-line tool which does the work of explaining errors messages for you. Simply run **splain**, paste in Perl's cryptic one-line error message, and **splain** will output the diagnotics that the diagnostics pragma would have given you.

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
2. A comment (what your program does)
3. The strict pragma
4. The warnings pragma

For example:

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

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

Scalars

You can find a selection of general-utility scalar subroutines in the `Scalar::Util` module.

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.

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 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/perldoc?Attribute::Types). `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.

If you want to understand how Perl handles numbers, read **perldoc perlnumber**. If you're going to be dealing with really big (bigger than 2.4 billion) numbers have a look at the `bigint` pragma for big integers and `bignum` for big numbers (integers and floating point numbers). If you need big rational numbers for very precise calculations see the `bigrat` pragma. These are much slower than using your machine's native number support but will not be subject to integer or floating point overflow and, in the case of `bigrat`, rounding.

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: `\\`

Be careful, there’s a common syntax error when the last character of a string is a backslash:

```
print ’This is a backslash: \’;
```

In this case Perl reads the backslash as an escape for the quote character, and thus our string does not terminate. In this case you must tell Perl that you don’t want this effect by escaping the backslash:

```
print ’This is a backslash: \\’;
```

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. Print 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`.

say (Perl 5.10.0+)

Perl 5.10.0 brought in a number of new features into the language. One of the most handy of these is `say`; print with a newline. `say` works *exactly* the same as `print` but it adds a newline as the last character printed. Thus the following two statements are identical.

```
use v5.10.0;
print "Hello World!\n";

say "Hello World!";
```

If you are using Perl version 5.10.0 or higher, you may use `say` in your exercises instead of `print`.

Arrays

You can find a selection of general-utility array subroutines in the `List::Util` module.

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

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

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

Initialising an array

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

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

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

Reading and changing array values

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

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

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

Array slices

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

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

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

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

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

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

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

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

Array interpolation

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

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

Counting backwards

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

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

Finding out the size of an array

So if we don’t know how many items there are in an array, how can we find out? There are two ways you might do this.

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
```

However, if you print either `$last` or `$#fruits` you'll find the value is `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`.

Unfortunately, `$#fruits` is easily confused with `#$fruits` (a comment!), and it can often cause off-by-one errors and other bugs. Thus it is generally considered a bad idea.

Fortunately, there's an easier way to find out the size of an array. If you evaluate the array in a scalar context Perl will give you the only scalar value that makes sense: the number of elements in the array.

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

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

Using qw// to populate arrays

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

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

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

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

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

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

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

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

Printing out the values in an array

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

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

The first method takes advantage of the fact that print takes a list of arguments to print and prints them out sequentially. The second uses `join()` which joins an array or list together by separating each element with the first argument. The third option uses double quote interpolation and a little bit of Perl magic to pick which character(s) to separate the words with (usually a space character).

A quick look at context

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

- scalar context, and
- list context

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

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

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

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

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

```
my @a = (1,2,3);
my @b = (4,5,6,7);
print @a + @b;
```

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

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

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

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

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

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

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

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

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

Exercises

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

You can find a selection of general-utility hash subroutines in the `Hash::Util` module.

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 %hair_colour_of = (
        Judy    => 'red',
        Andy    => 'black',
        Betty   => 'blue',
        Darryl  => 'brown',
        Frances => 'blonde',
);
```

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:

```
$monthdays{"January"} = 31;
$monthdays{February} = 28;
...
```

Changing hash values

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

```
$hair_colour_of{Betty} = ’purple’;      # Betty changed her hair colour
```

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" (assuming we include 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.
- Hash keys and/or values can be locked after hash creation using the `lock_keys` and `lock_values` subroutines from the `Hash::Util` module. This can be used to prevent accidental key creation or altering of values.

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***.

While it is true that traditional Perl hashes have no internal order, it is possible to keep insertion order by also storing the keys in an array. Doing this yourself can be error-prone so to make things easier, you can use the `Tie::IxHash` module which manages this work for you.

```
use Tie::IxHash;

my %hash;
tie (%hash, Tie::IxHash);
```

`Tie::IxHash` is available from CPAN, and you can read more at http://search.cpan.org/perldoc?Tie::IxHash

To understand how this module works you may want to read **perldoc perltie**.

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? e.g. `print %friends;` Look at the order of the elements.
7. What happens if you try to print out the hash inside double quotes? Do you understand why this happens?
8. What happens if you attempt to assign an array as a value into your hash?

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

Special variables

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

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

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

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

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

The special variable $_

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

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

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

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

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

@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`.

To save you from having to process command line switches by walking through `@ARGV` Perl comes with two standard modules to make your life easier: `Getopt::Std` and `Getopt::Long`. `Getopt::Std` enables simple, single-letter command line switches (such as `-i`), whereas `Getopt::Long` allows longer words, usually prefixed by `--` (such as `--help`).

When using `Getopt::Std` we recommend using `getopts` with a second argument.

```
use Getopt::Std;
my %opts;

getopts('iof:D:', \%opts);
```

`getopts` takes the expected command line arguments and a hash to populate with the results. Unexpected command line arguments cause a warning to be emitted. For boolean switches (`-i` and `-o` in this example) the hash will contain appropriate keys (`i` and `o`) with true values, if the switch was present, or false otherwise. For switches which take arguments (that is, they are followed by a full colon (`:`) in the list of options) the hash will contain keys (`f`, `D`) with the given argument as their value.

```
use Getopt::Std;
my %opts;

getopts('iof:D:', \%opts);

# Change directory if present:
if($opts{D}) {
        chdir $opts{D} or die "Can't chdir to $opts{D} - $!";
}
```

%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**.

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.

Exercises

1. The **printargs** script mentioned in the @ARGV example can be found in `exercises/printargs.pl`. Run this script now passing in some arguments.
2. Write a program which takes two arguments from the command line (a name and a favourite food) and then prints out a message detailing that name likes that food. An answer can be found in `exercises/answers/favouritefood.pl`
3. A user's home directory is stored in the environment variable `HOME` (Unix) or `HOMEPATH` (MS Windows). Print out your own home directory.

4. What other things can you find in `%ENV`? You can find an answer in `exercises/answers/env.pl`

Programming practices

When dealing with any program of more than a few lines, it’s important to pay some attention to your programming style. If your workplace already had a style guide, you should use that. Otherwise there are some recommended style guidelines in **perldoc perlstyle**. We also recommend the book *Perl Best Practices* by Damian Conway, which provides excellent coverage of good style choices. If your program is hard to read, it will be hard to get assistance from others.

Perltidy

Perltidy is a script indenter and reformatter. If you ever need to deal with unreadable code, perltidy forms a great starting point. You can download perltidy (http://perltidy.sourceforge.net) and run it with its defaults, or customise it to use your preferred bracing style, indentation width etc. Customisation is easy and there are a huge number of options. The defaults are usually a fine starting place. Perltidy can also be used to generate colourised HTML output of your code.

```
perltidy untidy.pl
```

takes code which might look like this:

```
#!/usr/bin/perl
use strict;use warnings;my%monthdays=(January=>31,February=>28,March=>31,April=>
30,May=>31,June=>30 ,July=>31 ,August=>31,September=>30,October=>31
,November=>30,December=>31,);my%pizza_prices=(small=>’$5.00’,medium
=>’$7.50’,large=>’$9.00’,);print$monthdays{January};
```

and creates a second file (`untidy.pl.tdy`) containing:

```
#!/usr/bin/perl
use strict;
use warnings;

my %monthdays = (
    January   => 31,
    February  => 28,
    March     => 31,
    April     => 30,
    May       => 31,
    June      => 30,
    July      => 31,
    August    => 31,
    September => 30,
    October   => 31,
    November  => 30,
    December  => 31,
);

my %pizza_prices = (
    small  => ’$5.00’,
    medium => ’$7.50’,
    large  => ’$9.00’,
);

print $monthdays{January};
```

Perltidy comes with a sensible set of defaults; but they may not be right for you. Fortunately you can use **tidyview** as a graphical user interface to preview perltidy's changes to your code and see which options suit you best. You can download tidyview from CPAN (http://search.cpan.org/perldoc?tidyview).

Perl Critic

Perl Critic is a framework for assessing the quality of your code. By default it uses the guidelines in *Perl Best Practices* but you can customise it to instead use guidelines based on your organisations coding standards.

To give it a go, you can upload your code to the perlcritic website (http://perlcritic.com/). Start with "gentle" and work your way up to "brutal", fixing issues as they are reported.

You can also download Perl::Critic (http://search.cpan.org/perldoc?Perl::Critic). This allows you to customise the policies and run perlcritic on the command line (or over code when it's being submitted to your revision control system).

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.
- The **perltidy** command can help tidy badly formatted code.
- The **Perl Critic** framework can provide automated code analysis and feedback on coding practices.

Chapter 7. Operators and functions

In this chapter...

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

What are operators and functions?

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

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

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

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

Operators

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

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

Arithmetic operators

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

Table 7-1. Arithmetic operators

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

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

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

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

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

For example:

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

String operators

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

Table 7-2. String operators

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

These can also be used as short cut operators:

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

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

Exercises

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

Advanced exercise

1. Read **perldoc -f sprintf** and use either `sprintf` or `printf` with a format string to print your answer to the first exercise above with two decimal points. eg `$629.00` instead of `$629`. Check that this works correctly for 17 widgets at $37.56 each.

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 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 course.

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 7-3. Context-sensitive functions

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

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

Some easy functions

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

String manipulation

Finding the length of a string

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

```
#!/usr/bin/perl
use strict;
use warnings;

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.

```
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.

```
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.

```
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`ing a single string. `chomp()` is invaluable for removing extraneous newlines from user input.

```
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.

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

# Notice that we substr and then print, when we're changing

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

When dealing with data in fixed-length formats, binary encoded integers, or floating points numbers you may find `pack` and `unpack` useful. See **perldoc -f pack** for more information. A tutorial can be found in **perldoc perlpacktut** and another online on Perlmonks (http://perlmonks.org/?node_id=224666).

Exercises

1. Create a 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

push, pop, shift and unshift

To add an element to the end of a list, when we don't know how long the list is; we can use `push`. This is extremely useful when walking through data and building up a list of interesting values, or for generating data on the fly.

```
# Build an array of 10 random numbers

my @random_numbers;
foreach (1..10) {
        push @random_numbers, int(rand(10));
}
```

`pop` allows us to retrieve the last item from a list (removing it at the same time).

```
my @stack = qw(a b c d e f g);

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

`unshift` and `shift` act like `push` and `pop` respectively, but at the start of the list.

```
my @array = (1 .. 5);

my $first = shift @array;   # $first is '1', @array is 2, 3, 4, 5

unshift @array, 10;         # @array is 10, 2, 3, 4, 5
```

Using these functions we can implement the common structures of queues and stacks in Perl.

`push` and `unshift` can be used to add multiple elements to the list at the same time:

```
my @array = (1..5);

push @array, 6..10;             # @array is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

unshift @array, 20, 30, 40;     # @array is 20, 30, 40, 1 .. 10
```

If you want to add or remove items in the middle of an array you'll need to use `splice`. It works very similarly to `substr`. Check out **perldoc -f splice** and page 793 of the Camel book (page 219, 2nd Ed) for more information.

Ordering lists

The `sort()` function, when called in list context, returns a sorted version of its argument list. It *does not* alter the original list.

The `reverse()` function, when called in list context, returns its argument list in reverse order. It *does not* alter the original list.

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

By default the `sort` function sorts ascii-betically (that is according to the ASCII table). Thus `A` preceeds `a`, but so does `B` and all the other upper case characters. Further, while `1` preceeds `2` so does `10`. We can change the default sort order by giving `sort` a block as its first argument:

```
my @list = ("a", "z", "c", "m");
my @numbers = (3, 5, 1, 9, 3, 1, 11, 2, 0);

# Sort ascii-betically (same as the default)
# Note that there is no comma between the block and the list
@list = sort { $a cmp $b } @list;

# Sort numerically
@numbers = sort { $a <=> $b } @numbers;
```

In the sort block, `$a` and `$b` are special arguments which represent the two values currently being compared. `cmp` and <=> are comparison operators which return -1, 0 or 1 depending on whether the value on the left is less than, equal to or greater than the value on the right. These two operators are covered further in the conditionals chapter.

Converting strings to lists, and vice versa (split and join)

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.

```
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 detail later.

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

Exercises

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

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

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

Hash processing

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

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

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

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

Reading and writing files

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

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

Time

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

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

To convert date and time information into a human-readable string you may want to use `strftime` from the `POSIX` module:

```
use POSIX qw(strftime);

# Current time in YYYY-MM-DD format:
print strftime( "%Y-%m-%d", localtime() );
```

For information on what the format string identifiers mean, consult your system's `strftime()` documentation, for example **man strftime** on a *nix system. Also read the `strftime` documentation in **perldoc POSIX** for portability considerations. We also list the more portable format options in our strftime perl-tip (http://perltraining.com.au/tips/2009-02-26.html).

The `DateTime` framework provides a powerful and extensible mechanism for date and time handling in Perl. You can read its documentation and download it from the CPAN (http://search.cpan.org/perldoc?DateTime).

Exercises

1. Create a hash and print the keys from it. Now `delete` an element and print the keys again to verify that it's gone. (Answer: `exercises/answers/hash2.pl`)
2. Print out the date for a week ago (the answer's in `exercises/answers/lastweek.pl`)
3. Read **perldoc -f localtime**.

Chapter summary

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

Chapter 8. Conditional constructs

In this chapter...

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

What is a conditional statement?

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

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

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

What is truth?

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

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

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

The if conditional construct

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

The `if` construct looks like this:

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

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

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

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

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

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

```
# make sure we have apples              # 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 (useful) block is a single statement, for instance:

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

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

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

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

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

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

Scope

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

```
use strict;

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

print $a;                       # prints foo
```

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

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

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

print "$a, $b\n";           # prints "5, 0"
```

Temporary changes to Perl's special variables can be performed by using `local`. Localising and then changing our variables changes their value not only for the the block we've localised 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. Using local is important as it ensures that no matter how we leave the enclosing block, the variable will have its value reset appropriately.

Under Perl 5.10 and above, it is possible to use `my` to lexicalise `$_`. This means changes to `$_` can only be seen within the enclosing block (and any enclosed blocks) rather than by all functions and subroutines called in that block.

It is not possible to use `local` on variables declared with `my`, although you can localise individual array and hash elements.

```
$_ = "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"
```

Comparison operators

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

Table 8-1. Numerical comparison operators

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

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

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

Table 8-2. String comparison operators

Operator	Example	Meaning
`eq`	`$a eq $b`	Equality
`ne`	`$a ne $b`	Inequality
`lt`	`$a lt $b`	Less than (in "asciibetical" order)

Operator	Example	Meaning
gt	$a gt $b	Greater than
le	$a le $b	Less than or equal to
ge	$a ge $b	Greater than or equal to
cmp	$a cmp $b	String equivalent of <=>

Some examples:

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

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

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

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

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

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

Exercises

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

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

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

   c. Otherwise, print "Just right"

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

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

Existence and definitiveness

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

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

```
my $skippy;                             # $skippy is undefined and false

$skippy = "bush kangaroo";              # true and defined
print "true"    if $skippy;             # prints true
print "defined" if defined($skippy);    # prints defined

$skippy = "";                           # false and defined
print "true"    if $skippy;             # does not print
print "defined" if defined($skippy);    # prints defined

$skippy = undef;                        # false and undefined
print "true"    if $skippy;             # does not print
print "defined" if defined($skippy);    # does not print
```

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

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

if( exists $miscellany{"Blinky Bill"} ) {
        print "Blinky Bill exists.\n";          # Does NOT get printed
}

if ( exists $miscellany{koala} ) {
        print "koala exists\n";                 # This IS printed
}

if ( defined $miscellany{koala} ) {
        print "koala is defined\n";             # Does NOT get printed
}
```

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

The `exists` function is described in the Camel book on page 710 (page 164, 2nd Ed), and also by **perldoc -f exists**.

Exercise

The following exercise uses the hash below:

```
my %num_of_cars = (
        Bob     => 1,           # Bob has 1 car
        Jane    => 2,           # Jane has 2 cars
        Jack    => 0,           # Jack doesn’t own a car
        Paul    => undef,       # Paul didn’t answer the question
);
# Andrew wasn’t asked (he’s not in the hash)
```

You can find a starting hash in `exercises/friends.pl`.

1. Write a program which takes the name of a friend on the command line and returns the number of cars that friend has. You’ll want to produce different messages for the following cases: the friend has 1 or more cars, the friend has no car, the friend didn’t answer the question, the given person isn’t our friend.

   Remember we want to be able to add more friends into our hash later, without having to change the code. An answer can be found in `exercises/answers/exists.pl`

In the above exercise you may have tried to cover both friends with one car or many cars printing a phrase such as `car(s)`. Perl has the `Lingua::EN::Inflect` module for pluralising English words:

```
use Lingua::EN::Inflect qw(PL);

my $num_cars = $num_of_cars{$friend};

# Prints ’Jane has 2 cars’, and ’Bob has 1 car’.
print qq{$friend has $num_cars }, PL(’car’, $num_cars), qq{\n};
```

`Lingua::EN::Inflect` can do a lot more, and is aware of irregular nouns (eg, mouse and mice). You can read more about `Lingua::EN::Inflect` on its CPAN page (http://search.cpan.org/perldoc?Lingua::EN::Inflect).

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 confusion. If in doubt, use parentheses to force evaluation order.

If you really want to know about the precedence, Perl’s `&&` and `||` operators have a high precedence, and are evaluted early in most expressions. This makes them useful for when you wish to calculate a boolean result and assign it to a variable. The `and` and `or` operators have very low precedence, making them more useful in some of the constructs we’ll see later in this chapter. For more information on operator precedence, read **perldoc perlop**.

Table 8-3. Boolean logic operators

English-like	C-style	Example	Result

`and`	`&&`	`$a && $b`
		`$a and $b`
			True if both `$a` and `$b` are true; acts on `$a` then if `$a` is true, goes on to act on `$b`.

`or`	`||`	`$a || $b`
		`$a or $b`
			True if either of `$a` and `$b` are true; acts on `$a` then if `$a` is false, goes on to act on `$b`.

`not`	`!`	`! $a`
		`not $a`
			True if `$a` is false. False if `$a` is true.

Here’s how you can use them to combine conditions in tests:

```
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 ((false) or true)
```

Logic operators and short circuiting

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(my $in_fh, "<", "input.txt") || die("Can’t open input file: $!");
# Or
open(my $in_fh, "<", "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";
}
```

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

Perl’s `||` operator works in the following way:

```
$x = $a || $b;

# $x = $a, if $a is true
# $x = $b, otherwise
```

Essentially, the `||` operator returns the first true value from a set of options, or the last option otherwise. This makes it useful for selecting between a set of alternatives:

```
# Take $a if true, else $b, else $c, else 0.

$x = $a || $b || $c || 0;
```

In Perl 5.10.0 and later, it’s also possible to use the *defined-or* operator, `//`:

```
# Take $a if defined, else $b, else $c, else 0.

$x = $a // $b // $c // 0;
```

The *defined-or* operator is particularly useful when selecting between a set of alternatives where values such as `0` or the empty string are valid data points.

Boolean logic operators are a common way of supplying a default value to a variable:

```
# If $name was false, make it equal to "Anonymous"
$name ||= "Anonymous";               # $name = $name || "Anonymous"

# If input was undefined, make it zero.
$input //= 0;                        # $input = $input // 0;
```

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
}

# Count from 1 to 10
my $count = 1;
while ($count <= 10) {
        print "$count\n";
        $count++;
}
```

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

```
# Count down from 10 to 1
my $count = 10;
until ($count < 1) {
        print "$count\n";
        $count--;
}
print "Blast off!\n";
```

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

```
my $count = 0;
print "$count\n" while(++$count <= 10);

$count = 11;
print "$count\n" until($count-- < 1);
```

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

for and foreach

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

```
for ( initialisation; conditional statement; increment ) {
        BLOCK
}

for (my $count = 1; $count <= 10; $count++) {
        print "$count\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 my $value (@array) {
        print "$value\n";
}
```

Instead of explicitly naming a variable to represent our element, we can use the special variable `$_`. This is only recommended if your use of `$_` is limited to a single statement.

```
foreach (@array) {
        print "$_\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.

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

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

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

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

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

⚠ 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 my $value (@numbers) {
        $value  = $value * 2;
}
```

☞ You can give `foreach` more than one list inside the parentheses:

```
# Iterate over @list1 and then @list2
foreach (@list1, @list2) {       }

# Iterate over some items
foreach( @array, 1..5, ’a’..’z’, $a, $b, $c, qw(Polly Rodger Sam) ) {
        ...
}
```

Exercises

1. Print out the keys and values for each item, from a hash, 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 `for` loop to print out a numbered list of the elements in an array
3. Now do it with a `while` 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:

```
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.

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

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

The above construct is exactly equivalent to our previous example.

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

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

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

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

Exercises

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

You'll have to type some stuff in, line by line. When you've finished entering input, hit **CTRL-D** (a.k.a. `^D`) on Unix or **CTRL-Z** (a.k.a. `^Z`) for Windows. This character sequence stands for *end of file* (`EOF`) which is false. Thus the `while` loop will end and any further code will be executed.

1. Try running the script with no arguments.
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:

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

By convention, 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`.

```
while (my $input = <STDIN>) {
        chomp $input;               # remove newline

        # skip blank lines
        if($input eq "") {
                next;
        }
        # quit if given "q" or "Q"
        elsif(lc $input eq "q") {
                last;
        }

        # Do something with $input...
}
```

Writing `next` tells Perl to repeat the smallest enclosing loop after re-evaluating the conditional. Writing `last` tells Perl to jump to the end of the smallest enclosing block and continue program execution from there.

By default `next` and `last` affect the current smallest loop. If we want to affect a different loop, naming our blocks become useful, as both `next` and `last` can take an argument specifying the name of the block they apply to:

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

        # we split the line into words and check all of them
        foreach my $word (split /\s+/, $input) {
                last LINE if lc($word) 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:

```
foreach my $file (@files_to_delete) {
        print "Are you sure you want to delete $file? [y|n]\n";

        my $answer = <STDIN>;
        chomp $answer;
        $answer = lc $answer;

        if($answer eq 'y') {
                unlink $file or warn "Delete failed: $!";
        }
        elsif($answer eq 'n') {
                print "$file not deleted\n";
        }
        else {
                # invalid input
                redo;
        }
}
```

In this case, had we used `next` the file we were dealing with would have been lost when we re-evaluated the condition <>. Thus the file would be neither deleted, nor reported on as not deleted. `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.

For very easy (and impressive) user interaction, check out the `IO::Prompt` module on CPAN (http://search.cpan.org/perldoc?IO::Prompt).

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.

Smart-match

In Perl version 5.10 (and above) there is a new operator called *smart-match*. This allows us to compare elements which aren't necessarily of the same type. Perl 5.10 also includes the `say` function which is the same as `print` but which always ends with a newline. We can use `say` through the `feature` pragma.

```
use feature qw(say);

my $x = "foo";
my $y = "bar";
my $z = "foo";

say ’$x and $y are identical strings’ if $x ~~ $y;
say ’$x and $z are identical strings’ if $x ~~ $z;    # Printed
```

As well as comparsion for equality, we can compare the value against a regular expression to see if that returns true:

```
my $input  = <STDIN>;

say ’You said the secret word!’ if $input ~~ /xyzzy/;
```

If we compare an array against a scalar, the result is true if the scalar is in that array. If we compare two arrays the result is true if both arrays are identical. If we compare an array to a regular expression, the result is true if any element in the array matches that expression.

```
my @foo = qw(x y z xyzzy ninja);
my @bar = qw(x y z xyzzy ninja);

say "Array contains a ninja " if ’ninja’ ~~ @foo;

say "Identical arrays"        if @foo ~~ @bar;

say "There is magic"          if /xyz/ ~~ @foo;
```

Likewise, if we compare a scalar against hash, the result is true if the scalar is a key in that hash:

```
my %colour = (
        sky   => ’blue’,
        grass => ’green’,
        apple => ’red’,
);

my $input = <STDIN>;
chomp $input;

say "I know the colour of $input" if $input ~~ %colour;
say "A key starts with ’gr’"      if /^gr/  ~~ %colour;
```

Note well: in Perl 5.10.0 the ordering of operands to smart-match did not matter. In 5.10.1 they do matter, with the rightmost argument primarily determining behaviour.

given and when

Perl version 5.10 introduced the new `given` and `when` keywords. This gives Perl a native switch statement. To enable it we use the version of Perl we wish to have the functionality from:

```
use v5.10.0;
```

We can solve the above guessing game using `given` and `when` as below.

```
use strict;
use warnings;
use v5.10.0;

# Pick our random number between 1 and 100
my $secret = int(rand 100)+1;

# An array of numbers guessed thus far
my @guessed;

say "Guess my number between 1-100";

# Get each guess from the user
while (my $guess = <STDIN>) {
        chomp $guess;

        # Check their guess using given/when
        given($guess) {
                # See if there are any non-digits
                when (/\D/)         { say "Give me an integer"; }

                # Check if the number is in our @guessed array
                when (@guessed)     { say "You’ve tried that";  }
                # Check if the number is the same as $secret
                when ($secret)      { say "Just right!";  last; }

                # Else it’s too low or too high
                when ($_ < $secret) { say "Too low";  continue; }
                when ($_ > $secret) { say "Too high"; continue; }

                # record the guess they’ve made
                push(@guessed,$_);
        }
}
say "You took ", scalar(@guessed), " guesses.";
```

The `given` keyword marks the start of our switch logic. It has the effect of assigning the contents of `$guess` to `$_` to for the duration of the block. However, this change to `$_` is not visibile outside the `given` construct - it is lexical. Each `when` is a case statement and each of these are checked until there is a match. These use smart-match underneath against the `$_` value.

`when` blocks have a conceptual `break` after each one, thus once a match is found no other part of the `given` structure is checked or executed. To over-ride this behaviour, we use `continue` in our last two cases to ensure that we can push the (valid) guess the user made onto our `@guessed` array.

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

For a more comprehensive coverage than we give in this chapter, read **perldoc perlsub**.

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";
        return;
}
```

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 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 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";
        return;
}
```

You can take any number of scalars in as arguments - they'll all end up in `@_` in the same order you gave them. Keep in mind that we have to unpack `@_` in the same way for one variable as for more than one:

```
send_email($client_address);
send_email($manager_address);
send_email($personal_address);

sub send_email {
        my ($address) = @_;

        # ...
}
```

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";
        return;
}
```

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

Passing in arrays and hashes

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

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

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

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

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

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

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

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

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

two_lists(@chosen, @colours);

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

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

        ...

        return;
}
```

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:

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

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

        my @chosen_copy  = @$chosen;
        my @colours_copy = @$colours;

        # at this point @chosen_copy contains: (red white green)
        # and @colours_copy contains (red blue white green pink)

        ...

        return;
}
```

however these are beyond the scope of this section.

References will be covered in more depth later in the course. To learn more about them now 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 some random integers between 0 and $max
sub random_ints {
        my ($how_many, $max) = @_;

        my @randoms;
        foreach (1..$how_many) {
                push @randoms, int(rand $max);
        }

        return @randoms;
}

my @three_rands = random_ints(3, 10);
# sets @three_rands to something like: (4, 7, 1);

# alternately:
my ($w, $x, $y, $z) = random_ints(4, 15);  # eg: $w=0, $x=3, $y=13, $z=2
```

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 -f wantarray**.

Exercises

1. One international foot is 0.3048 metres. Write a subroutine (feet_to_metres) that takes a length in feet, and `return`s the length in metres. Call this subroutine in your code and verify that it returns what you expect.

   ```
   print feet_to_metres(1), "\n";          # Should print 0.3048
   print feet_to_metres(2), "\n";          # Should print 0.6096
   ```

2. Use a loop to call your feet_to_metres subroutine for lengths of 1 foot to 10 feet, to display their equivalent lengths in metres.
3. You have been hired by the mayor’s office to develop a system to contact superheroes. Write a subroutine that returns a standard letter and which accepts three arguments: a superhero to contact, the location they are required, and the threat they must combat. Use your subroutine to generate a letter asking Batman to save Gotham City from The Joker.
4. Write a subroutine that takes a list of numbers, and calculates and returns their mean (the sum of all numbers divided by the count of numbers). Use your subroutine to calculate the mean of the numbers 2,3,5,7,11,13,17,19.

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. 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";

        return "${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";
}
```

Subroutine declaration and prototypes

Many programming languages allow or require you to predeclare your subroutines/functions. These declarations, also called prototypes, tell the compiler what types of arguments the subroutine is expecting. Should the subroutine then be passed too few, too many or the wrong kind of arguments; a compile-time error is generated and the program does not run.

While prototypes in Perl do exist, they are not the same as the above mentioned function declarations. Prototypes allow developers to write subroutines which mimic Perl’s built-in functions, but they don’t work the same was as they do in other languages. When used with regular subroutines, the consequences can be surprising and difficult to understand.

It is recommended that you avoid using Perl’s subroutines prototypes.

Should you have a requirement to validate your subroutine parameters the `Params::Validate` module, available from CPAN, will do all that you want and more.

Chapter summary

- Parameters in Perl are usually passed "by position".
- Positional parameter passing makes having independent optional arguments or extra arguments difficult.
- Using positional parameter passing requires the programmer to remember or look up the parameter order when dealing with subroutines that take many arguments.
- Named parameter passing makes independent optional arguments and extra arguments easy.
- Named parameter passing allows the programmer to list the arguments in an easy to understand and change manner.
- Using named parameter passing, it becomes very easy to create default values for parameters.

Chapter 11. 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. One of Perl's most common uses is string processing and it excels at that because of its built-in 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.
- The replacement string is treated as if it were a double quoted string.

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

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 11-1. Binding operators

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

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

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

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

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

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

Easy modifiers

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

Table 11-2. Regexp modifiers

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

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

Meta characters

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

Some easy meta characters

Table 11-3. Regular expression meta characters

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

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.

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

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

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

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
```

☞ You'll notice that in the last expression, we have a lot of backslashes. Some are in front of letters and some in front of punctuation. The rule to remember is: if there's a backslash in front of a letter, that means it's a meta character and means something special. Backslash in front of punctuation that means it's the literal punctuation character - in other words, it *doesn't* mean anything special.

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 11-4. 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 four words (without punctuation)
/(\b\w+\s*){4}/;

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

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

Exercises

1. Determine whether your name appears in a string (an answer is in `exercises/answers/namere.pl`).
2. What pattern could be used to match a blank line? (Answer: `exercises/answers/blanklinere.pl`)
3. 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).
4. Write a script to search a file for any of the names "Yasser Arafat", "Boris Yeltsin" or "Paul Keating". Print out any lines which contain these names. You can find a file including these names and others in `exercises/famous_people.txt`. (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. You can find a test file in `exercises/elvis.txt`. (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 12. Introduction to Modules

In this chapter...

In this chapter we'll discuss modules from a user's standpoint. We'll find out what a module is, how they are named, and how to use them in our work.

What is a module?

A module is a separate file containing Perl source code, which is loaded and executed at compile time. This means that when you write:

```
use CGI;
```

Perl looks for a file called `CGI.pm` (**.pm** for *Perl Module*), and upon finding it, loads it in and executes the code inside it, before looking at the rest of your program.

Module uses

Perl modules can do just about anything. In general, however, there are three main uses for modules:

- Changing how the rest of your program is interpreted.

  For example, to enforce good coding practices (`use strict`) or to allow you to write in other languages, such as Latin (`use Lingua::Romana::Perligata`).

- To provide extra functions to do your work.

  For example `use Carp`, `use File::Copy` and `use Data::Dumper`.

- To make available new classes for object oriented programming.

  For example `use Moose`, `use WWW::Mechanize`.

Sometimes the boundaries are a little blurred. For example the `CGI` module provides both a class and the option of importing subroutines, depending on how you load it.

use statements

When compiling your script, Perl first does a full pass over your code to find all the `use` statements. These are then each loaded and compiled in turn. Thus in the following example `File::Copy` is still loaded at compile time even though it is inside a conditional:

```
if( $interesting ) {
        use File::Copy qw(copy);
        copy($filea, $fileb) or die "Failed to copy: $!";
}

# Still using File::Copy's copy out here.
copy($fileb, $filea) or die "Failed to copy: $!";
```

Thus it is always a good idea to `use` your modules at the top of your program.

If you want to load a module depending on some condition, you can use the `if` pragma. See `perldoc if` for more information.

Lexically scoped modules

Some modules (most often pragmas, for example `strict`, `warnings` and `autodie`) are lexically scoped. These are still loaded during compile time, but they only affect the scope that they are `used` within. For example;

```
if( $interesting ) {
        use warnings;   # warnings turned on for this block

        print undef;    # warns
}

# warnings not turned on out here
print undef;        # no warning
```

Lexical modules can have their effects turned of by preceding them with `no`:

```
use strict;
use warnings;

# Now I’m going to do something silly
if( $silly ) {
        no strict;
        no warnings;

        # silly thing
}
```

Passing in directives

When a module is *use*d, its `import` subroutine (if one exists) gets called with any directives specified on the *use* line. For example:

```
use File::Copy qw(copy);
```

This is useful if you want to export functions or variables to the program using your module. Note the use of `qw()`, this is a list of words (in our case, just a single word). It’s possible to pass many options to a module when you load it. In the case above, we’re asking the `File::Copy` module for just the `copy` subroutine.

Each module has a different set of options (if any) that it will accept. You need to check the documentation of the module you’re dealing with to find out which options will be useful for you.

To find out what options exist on any given module read its documentation: `perldoc` *module_name*.

The double-colon

Most Perl modules contain one or more double-colons in their names, for example `File::Copy`, `Data::Dumper` and `Finance::Quote`. The double-colon allows us to group similar modules together, in the same way that we group together similar files.

In fact, the file analogy is so true, that when Perl searches for a module it converts all the double-colons into your directory separator and uses that to find the appropriate file to load.

```
File::Copy          =>  File/Copy.pm
File::Spec          =>  File/Spec.pm
Data::Dumper        =>  Data/Dumper.pm
Finance::Quote      =>  Finance/Quote.pm
```

Just because two modules live in the same directory, such as `File::Copy` and `File::Spec`, doesn't necessarily mean that their code is related. In this case both modules are useful in interacting with the file system, and `File::Copy` relies on `File::Spec` but that's it.

Typically modules are named from the most general category to the most specific. It's perfectly legal to have many double-colon separators in modules names so `Chicken::Bantam::SoftFeather::Pekin` is a perfectly valid module name, if a little unlikely.

Exercise

1. Use `File::Copy` to make a copy of one of your files (for example `elvis.txt`). Verify that it worked, don't forget to handle errors.
2. Prompt the user to give you the name of a file to copy, and the new file name.

The Comprehensive Perl Archive Network (CPAN)

One of Perl's biggest strengths is the Comprehensive Archive Network (also known as CPAN). The most commonly used portal into this is http://search.cpan.org. It is highly recommended that you become familiar with searching CPAN as many common problems have been solved and the code placed in modules on the CPAN.

Exercise

1. Open a web browser to http://search.cpan.org, CPAN's search site, and spend a few minutes browsing the categories provided.
2. Perform a search on CPAN for a problem domain of your choice. If you can't think of one, search for `XML`, `SOAP` or `WWW`.

CPAN features

Each module has a wealth of information available about it. You can see the module's source code, browse the files included with it (which hopefully include tests and example usage), see bug reports, regularity of releases as well as the date of last release, see annotations by other users, determine the likelihood of successful installation on your system and much more. As you become more familiar with CPAN many of these extra features will become invaluable.

Using CPAN modules

The CPAN is a vast repository of freely available modules. This makes CPAN an excellent starting point for any project you might want to start. You should keep in mind, however, that not all modules are created equal. Some modules (such as `DBI` and now also `DBIx::Class`) have become de-facto standards, while others may not be used by anyone at all, not even their author.

As with any situation where you're using third party code, you should always take the time to determine its suitability for the task at hand. Fortunately, in almost all circumstances, it's better to use or extend a suitable module from CPAN than it is to try to re-invent the wheel yourself.

Installing CPAN modules to the system directories

For coverage on installing modules on various operating systems read `perldoc perlmodlib`. If you want to distribute your own modules read `perldoc perlnewmod`.

Many of the popular CPAN modules are pre-packaged for popular operating systems. Thus, on a Unix-flavoured operating system you may just need to use your build in package manager, or ask your systems administrator. If the module has not been pre-packaged you can install it manually.

Installing manually from a tar ball

If you have obtained a `.tar.gz` or `.tgz` file containing your module (such as if you have downloaded it directly from CPAN) you can install it manually as follows:

First un-tar the module and change into that directory. In Unix we might write:

```
tar -xvzf some-module-0.0.tgz
cd some-module-0.0
```

Whereas in Microsoft Windows we might use WinZip or another archive management tool.

Then, from a command prompt run:

```
perl Makefile.PL
make
make test
make install
```

This will install the module into your system directories if you have permission to do so (such as if you run `make install` as a superuser on Unix).

As a general rule, Microsoft Windows systems don't include `make` or a C compiler and these will need to be installed first. Once these have been installed, you may need to run the previous commands with `nmake` or `dmake` instead of `make`.

Installing manually with PPM (ActivePerl on Windows)

If you are using Active State's Active Perl (which is a common choice with Microsoft Windows) you can use the PPM (Programmer's Package Manager) to download and install modules. These have been pre-compiled for Windows and thus do not require a C compiler. Not all CPAN modules are available by PPM and not all PPM distributions are as up to date as CPAN.

Installing manually with cpan (Unix and Windows)

Installing modules with the `cpan` shell that comes standard with Perl is easy. At your command prompt you would write:

```
cpan Text::CSV_XS
```

to install the Text::CSV_XS module.

Installing manually with cpanminus

While installing modules with `cpan` is generally easy, it is very verbose and can be a huge hassle if you are missing modules that are in the dependency tree. You have to wait for prompts asking if you'd like to install those modules too; and sometimes, due to dependency resolution ordering, you'll have to re-run the command to finish the install.

A neat alternative is `App::cpanminus`, which doesn't come standard with Perl, so you'll have to install it yourself, which is as easy as as saving the source code at http://cpanmin.us/ to a file named `cpanm`.

`cpanminus` does auto-configuration, will automatically assume you want all required dependencies and tries to be as quiet as possible. Once you've installed it, all you need to write is:

```
cpanm Text::CSV_XS
```

and it'll install `Text::CSV_XS` and any dependencies for you.

Exercise

1. Using `cpan`, install `IPC::System::Simple`. If it is already installed, install `Text::CSV_XS`.

Finding installed modules

The Camel book describes the standard modules in chapters 31 and 32. (Chapter 7, 2nd Ed.)

Perl comes with many modules in its standard distribution which are installed by default. Additionally, modules may be installed manually or by your system's package manager as discussed above. The following commands may be useful for finding out which modules are on your system:

perldoc perlmodlib

Gives a list of all the standard libraries for your version of Perl, along with a very brief description for each.

perldoc perllocal

Gives a list of all the modules that have been installed manually (and extra operating specific modules included with your distribution that are not part of the standard library. Modules installed through your operating system's package manager may not appear in this list.

perldoc -q installed

To find a complete list of modules available on your system, regardless of how they were installed, read the documentation in the faq above.

There are some other commands that are very useful when working with modules.

perldoc *modulename*

This gives the documentation for the specified module. For example `perldoc File::Copy` will give you information about the `File::Copy` module.

perldoc -l *modulename*

This gives the file system location for the module for your version of Perl. If you are finding that a module doesn't quite behave the way you expect, this can be used to check whether you're going to load the version you thought you would (although `use lib` lines will change this).

perldoc -m *modulename*

If you're curious about how a module works under the hood, this command will allow you to view the source code of that module.

How does Perl find the modules?

Perl searches through a list of directories that were determined when Perl was compiled (this list is different for each operating system). You can see this list (and all the other options Perl was compiled with), by using `perl -V` and looking at what is in `@INC`. For example (on an Ubuntu machine) you might have:

```
@INC:
  /etc/perl
  /usr/local/lib/perl/5.10.0
  /usr/local/share/perl/5.10.0
  /usr/lib/perl5
  /usr/share/perl5
  /usr/lib/perl/5.10
  /usr/share/perl/5.10
  /usr/local/lib/site_perl
```

`@INC` contains the include path for Perl (all the directories Perl will look in for modules).

Adding paths to @INC

There are a number of ways to change `@INC` to add our own paths as well. This is especially important when we've installed our own private copies of modules.

use lib

The `lib` pragma allows us to inform Perl about extra directories. For example:

```
use lib "/path/to/libs";
use Module;         # Perl will look in /path/to/libs first
```

-I

If we're calling a program from the command line and wish to add a new include path for this invocation we can use the `-I` switch to Perl. For example:

```
perl -I /path/to/libs program.pl
```

Now Perl will look in /path/to/libs first for each module we use.

PERL5LIB

If we set the `PERL5LIB` environment variable to contain one or more directories (colon separated) then each of these will be added to `@INC`. Note that if your script is running with taint checking enabled this environment variable will be ignored.

```
export PERL5LIB="/path/to/libs/";
perl program.pl
```

Now Perl will look in `/path/to/libs` first for each module we use.

Since `use` lines are evaluated before regular Perl code, modifying `@INC` directly will not have the desired effect.

You can use the `lib` pragma with directory paths that don't exist without causing any issues:

```
use lib "c:/myperl/projectA/lib";
use lib "/usr/local/src/projectA/lib";
```

Relative paths

Typically our programs are called from the same directory they are stored in:

```
# cd /home/sue/scripts/
# ./program.pl
```

When we expect this to be how our programs are called, we can specify include paths relative to our program code:

```
    use lib 'some/relative/path';
```

and this will correctly use the directory `/home/sue/scripts/some/relative/path` and so our modules will be found. If, however, our program is called from a different directory (thus making our current working directory something else) this will not work.

```
    # cd /tmp/
    # /home/sue/scripts/program.pl
```

In the above example, our current working directory is `/tmp` and all relative paths will be from `/tmp`, so the above `use lib` line will look for modules in `/tmp/some/relative/path/`.

If it is possible that your script will be invoked from a different directory in which it resides, you can use `FindBin` to provide the location of the Perl program and use that. (`FindBin` gives us this directory in a scalar `$Bin`).

```
use FindBin qw($Bin);

# use the current directory (relative to script) as a library path
use lib "$Bin/";

# use our (relative) p5lib directory as a library path
use lib "$Bin/p5lib";
```

`FindBin` can also be used to get the basename of the Perl script (`$Script`), a canonical path (with all links resolved) (`$RealBin`) and the script name plus canonical path (`$RealScript`).

To find other alternatives to dealing with relative paths, read `perldoc -q library`.

Working with a local install directory

So far we've covered how to install modules into Perl's standard system locations, and also how to add paths to `@INC` if there are modules we wish to use that are not in those standard locations. This is handy for two reasons. It gives us the ability to create our own modules and use them locally, and secondly it allows us to install modules into our home directories, or somewhere other than the standard locations.

Manually installing modules locally

To install a module locally when doing a manual install from a tarball, we pass in an extra command when we create the makefile:

```
perl Makefile.PL INSTALL_BASE=/home/username/perl5
```

The `INSTALL_BASE` directive specifies the general location for files to be installed. Our library files will end up in `INSTALL_BASE/lib/perl5`, any binaries and scripts the module installs end up in `INSTALL_BASE/bin`, and our Unix man files end up in `INSTALL_BASE/man/man1` and `INSTALL_BASE/man/man3`.

To use these modules, we'd need to add following lines to our code (the second line is architecture specific):

```
use lib "$ENV{HOME}/perl5/lib/perl5";
use lib "$ENV{HOME}/perl5/lib/perl5/i486-linux-gnu-thread-multi";
```

or set our `PERL5LIB` environment variable to include `'/perl5/lib/perl5;~/perl5/lib/perl5/i486-linux-gnu-thread-multi'`.

Installing modules locally with cpan

If you're using cpan to install your modules, you can instruct it to always use your local directory with the following configuration instructions. First, start the cpan shell:

```
cpan
```

then:

```
o conf makepl_arg "INSTALL_BASE=/home/jarich/perl5"
o conf commit
```

From here you use cpan just as before, except that now it will install your modules in the defined location, rather than in the system directories.

As above, to use these modules you'll need to add

```
use lib "$ENV{HOME}/perl5/lib/perl5";
use lib "$ENV{HOME}/perl5/lib/perl5/i486-linux-gnu-thread-multi";
```

to your code or set our PERL5LIB environment variable to include '/perl5/lib/perl5;~/perl5/lib/perl5/i486-linux-gnu-thread-multi'.

local::lib and cpanm

local::lib allows us to have a hierarchy of Perl modules installed in our home directory or elsewhere transparently. It does not come with Perl by default, but is easy to install and bootstrap even if you do not have superuser permissions.

local::lib primarily sets a number of environment variables (for example):

```
export PERL_MB_OPT='--install_base /home/username/perl5'
export PERL_MM_OPT='INSTALL_BASE=/home/username/perl5'
exporT PERL5LIB='/home/username/perl5/lib/perl5/i486-linux-[...]'
export PATH="/home/username/perl5/bin:$PATH"
```

This means that if we install modules into /home/username/perl5/ then in our code we can just write:

```
use local::lib;
```

instead of setting the full path. Alternately, if we copy these environment changes into our .bashrc file (or as appropriate) then we don't need to add anything to our code, Perl will use our modules from /home/username/perl5/ where they exist, otherwise the system installed ones.

cpanm

cpanm is aware of local::lib. Thus if we're installing modules with cpanm we can just write:

```
cpanm --local-lib Text::CSV_XS
```

and Text::CSV_XS will be installed into our preferred directory. If you've already configured your shell enviornment variables, then installing to your local::lib directory will be done by default.

Microsoft Windows

`local::lib` works on Microsoft Windows, but in order to set the environment variables permanently you'll need to edit the registry. Another module `App::local::lib::Win32Helper` can do this for you.

PAR

If you need to distribute your code to systems which aren't guaranteed to have all your required modules, and which may not even have Perl installed, PAR is the solution. PAR is the Perl Archiver and allows you to package up your code, all its data files, its modules and Perl itself if necessary and can be used to create a stand-alone executable. For more information see (http://search.cpan.org/perldoc?PAR) the PAR documentation.

You may also find the `--local-lib-contained` switch to `cpanm` to be helpful as when it examines a module's dependencies, it assumes no non-core modules are installed on the system.

Finding quality CPAN modules

There are a lot of modules on the CPAN, and sometimes deciding which module is the best task for a job can be a little challenging. Luckily, there is a wealth of information to help you.

Firstly, many CPAN modules have ratings. After doing a search, any module with a star rating next to it has reviews about that module that can be viewed. A module with a large number of high-rating reviews is an indication of a quality module.

The CPAN Testing Service (http://cpants.perl.org/) maintains some automatically generated metrics on common quality indicators such as packaging structure, test suites, and dependencies.

There are lists of community-recommended modules on the Perl 5 Wiki (http://www.perlfoundation.org/perl5/index.cgi?recommended_cpan_modules). These also include links to further resources.

Finally, online communities such as PerlMonks (http://perlmonks.org/), or local Perl Mongers chapters (http://pm.org/) can often provide personal module recommendations. PerlMonks in particular has an extensive archive of module discussions and tutorials.

Chapter summary

- A module is a file containing Perl source code which grants extra functionality to our code and is executed at compile time.
- We can pass in directives to a module when loading it to allow us to request specific functionality.
- The double colon in file names represents the directory separator on the file system.
- There is more than one way to install modules.
- Perl searches for modules in `@INC`. We can add to `@INC` with the `use lib` pragma.
- `local::lib` makes working with locally installed libraries easy.

Chapter 13. Using Perl objects

In this chapter...

While discussion of Object Oriented programming is beyond the scope of this course, a great many modules you may encounter while programming provide an object oriented interface. This chapter will teach you what you need to know to use these modules.

Perl Training Australia runs a two day course on Object Oriented Programming in Perl, for more information visit our website (http://www.perltraining.com.au/) or talk to your instructor during a break.

You may also want to look at **perldoc perlboot**, **perldoc perltoot**, **perldoc perltooc**, and **perldoc perlbot**.

Objects in brief

An *object* is a collection of data (attributes) and subroutines (methods) that have been bundled into a single item. Objects often represent real-world concepts, or logical constructs. For example, an *invoice* object may have attributes representing the date posted, date date, amount payable, GST, vendor, and so on. The invoice may have various methods that allow for payment to be made, and possibly a payment to allow the invoice to be disputed.

An object in Perl is a reference to a specially prepared data structure. This structure may be a hash, an array, a scalar or something more complex. However, as the user of an object, we don't need to know (and should not care) what sort of structure is actually being used. What matters are the *methods* on the object, and how we can use them.

A Perl object is a *special* kind of reference because it also knows what class it belongs to. In other words, an object knows what kind of object it is.

Object orientation allows us to create *multiple* objects from the same class which can each store different information and behave differently according to that information. This makes it very easy for the users of those objects, as it makes the information easy to track and manipulate.

Loading a class

To use a Perl module which provides an object oriented interface we `use` it without specifically importing any methods. For our examples we will use the `DBI` module, which allows us to interact with a number of databases, and is one of the most commonly used modules in Perl.

```
#!/usr/bin/perl
use strict;
use warnings;

use DBI;                # We can now create DBI objects.
```

To learn more about `DBI` read **perldoc DBI** and the `DBI` homepage (http://dbi.perl.org/).

Perl Training Australia also runs a Database Programming with Perl course which you may find of interest. For more information visit our website (http://www.perltraining.com.au/) or talk to your instructor during the break.

Instantiating an object

To create a new object we call the constructor method on the *name* of the class. In many cases this method is called `new`, however with `DBI` it is called `connect`; as we get our database handle by *connecting* to a database.

```
use DBI;
use WWW::Mechanize;

# Create a DBI object (database connection handle)
my $dbh = DBI->connect($data_source, $username, $password);

# Create a WWW::Mechanize object (agent)
my $mech = WWW::Mechanize->new();
```

By convention, our connected database object is called `$dbh`, for "database handle".

We can create a number of database handles (objects), with each connecting to different databases or with different usernames and passwords. We could also create a number of database handles connecting to the same database. This could potentially be useful if we wished to execute multiple SQL commands simultaneously, particularly if we’re dealing with a clustered database system.

```
use DBI;

my $oracle_dbh   = DBI->connect($oracle_dsn, $oracleuser, $oraclepasswd);
my $postgres_dbh = DBI->connect($postgres_dsn, $postgresuser, $postgrespasswd);
my $mysql_dbh1   = DBI->connect($mysql_dsn, $mysqluser1, $mysqlpasswd1);
my $mysql_dbh2   = DBI->connect($mysql_dsn, $mysqluser2, $mysqlpasswd2);
```

Each of these objects represent a different database connection and we can call the other `DBI` methods on these objects from now on. Each object will remember which database it refers to without further work on behalf of the programmer.

Calling methods on an object

We can get at the contents of a normal reference by using the arrow operator:

```
$array_ref->[$index];           # Access array element via array reference
$hash_ref->{$key};              # Access array element via hash reference
```

In the same way, we can access Perl object methods (or functions, if you’d prefer) the same way:

```
$object->method();              # Call method() on $object
```

In a specific case, we can call a method on one of our `DBI` objects as follows:

```
use DBI;

my $dbh = DBI->connect($data_source, $username, $password);

$dbh->do("UPDATE friends SET phone = ’12345678’ WHERE name = ’Jack’");
```

Destroying an object

When you no longer need an object you can let it go out of scope, just as when you no longer need any other Perl data structure. In some cases the documentation may recommend calling certain clean up functions. In the case of DBI it is considered polite to disconnect from the database.

```
$dbh->disconnect();
```

Chapter summary

- Perl objects are special references to Perl data structures which know which class they belong to.
- Object orientation allows us to create multiple objects from the same class to store different information.
- To use a Perl class we just use the module.
- To create an object we call the constructor method on the class.
- Many objects of the same class can be created.
- To call a method on an object we use the arrow operator.
- Objects are destroyed when they go out of scope.

Chapter 14. Testing your module

> A program is secure if you can depend on it to behave as you expect.

A module is not complete, until you are certain that it works correctly. The easiest way to ensure that this is the case, and to protect yourself from introducing bugs in the future, is to create a test suite for your code. Perl has a fantastic testing culture and many great tools to make testing easy.

For further information on testing read the `Test::Tutorial` at http://search.cpan.org/perldoc?Test::Tutorial.

Testing Strategies

Testing cannot prove the absence of bugs, although it can help identify them for elimination. It's impossible to write enough tests to prove that your program is flawless. However, a comprehensive test-plan with an appropriate arsenal of tests can assist you in your goal of making your program defect free.

When testing there are two typical paradigms for testing, these are as follow:

Black box testing

You have a specification, and you test that the code meets that specification. This doesn't require any knowledge of how the code works. For example, if a date could be input, you might try each of the following:

- Today's date
- Yesterday, tomorrow
- One year from now
- One year ago
- One hundred years from now
- One hundred, two hundred or three hundred years ago
- 29 February, on a leap year
- 29 February, not on a leap year
- 32nd January, any year
- 31st April, any year
- Partial dates
- Empty dates
- Non-dates
- Different date formats (20-01-2011 vs 20/01/2011).
- Different date arrangements (20-01-2011 vs 01-20-2011)

The specification should state which of the above are valid and invalid, and how each should be handled. In either case, so should the documentation. For example, your program may only allow for the date to be valid if it occurred in the last 150 years (in which case you'd test 149, 150 and 151 years ago, as well).

White box testing

You know how the code works, you need to test the edge cases. For example if you accept someone's address and then store it in a database field that accepts 120 characters then you'd check:

- A zero character address
- An address containing 1 character
- An address consisting of just 1 database meta-character, such as `'`
- An address of 119 characters
- An address of 120 characters
- An address of 121 characters
- An address containing 120 characters exactly but which also includes several meta-characters such as `'` which will need to be escaped for the database

You'd then verify that the data you retrieve in is exactly the same as the data you stored, or that an error is given as appropriate.

Combining these ideas

It should be clear to see that both a combination of white box and black box testing strategies is required to give us a robust testing suite. For example, in the white box list above, we're not testing that the address is an address, just that our code handles it correctly. If we were to add address validation, however, then we'd look carefully at what that validates and add extra tests to test that out (and also tests we expect to fail).

Our project

In your **`exercises/project/`** directory, you'll find some project files.

The `Client-Aggregate` directory contains the `Client::Aggregate` distribution as created by `module-starter`, including a testing directory `t/` which we'll be working with later in this course.

In the top level directory, the is a useful simple program `interrogate.pl`. This can be used to perform searches (using the module) on our client data.

There is also a data file `client_data.csv` which represents a flat file database of our client information. This data has been automatically generated and does not contain details of real people.

Our `Client::Aggregate` module provides subroutines which allows us to filter our clients on various attributes. the `Client::Record` module provides a way of interacting directly with a client record.

Exercise

Using the `interrogate.pl` program in your `exercises/project/` directory (and reading the help provided by running `interrogate.pl --help`) perform the following queries on your client data:

1. Who has client id 10?
2. Which clients live in postcode 4000? How about postcode 3000?
3. Which clients live in Victoria?
4. Which clients live in Victoria and are female?
5. Which clients live in Victoria, are female and have a name starting with "A"?
6. Which clients have a non-binary gender?

Running our test suite

Although we haven't added any tests to our project yet, we can run the tests that `module-starter` gave us. These can be found in our `t/` directory, so `Client-Aggregate/t` in this case. Let's look at what tests we have to start with:

```
00-load.t  boilerplate.t  manifest.t pod-coverage.t  pod.t
```

00-load.t

This tests that our module can be loaded without errors. This should be the first test run, hence starting with the number `00`.

boilerplate.t

This test warns the author against leaving boilerplate documentation in the README, Changes and lib/module files.

Initially these tests are marked in a TODO block, to prevent their failure from slowing you down.

manifest.t

This test checks that your manifest is up to date. The manifest contains the list of files your distribution contains, and a list of dependencies that your distribution relies on. This information is essential if you are going to distribute your module to CPAN.

pod.t

This tests whether your POD in your file is valid.

pod-coverage.t

This tests whether you appear to have provided sufficient documentation for your code. Every subroutine name which is exported must appear in a `<=head(n1)`>> or `=item` block.

Depending on your version of `Module::Starter` you may have additional tests as well.

We can run these tests via either of the following:

```
perl Makefile.PL
make test
```

```
# or

prove -l lib/ t/
```

If you are on a Windows machine you may need to use `dmake` as covered earlier.

In either case the output should look almost the same:

```
t/00-load.t ....... 1/2 # Testing Client::Aggregate 0.01, Perl 5.010001, /usr/bin/perl
t/00-load.t ....... ok
t/boilerplate.t ... ok
t/manifest.t ...... skipped: Author tests not required for installation
t/pod-coverage.t .. ok
t/pod.t ........... ok
All tests successful.

Test Summary Report
-------------------
t/boilerplate.t (Wstat: 0 Tests: 4 Failed: 0)
  TODO passed:   3-4
Files=5, Tests=10,  1 wallclock secs ( 0.11 usr  0.03 sys +  0.64 cusr  0.06 csys =  0.84 CPU)
Result: PASS
```

☞ When we run `perl Makefile.PL` it creates a `Makefile` for us. Amongst other things, this records the names of the tests that exist in our distribution. If you have not *added* or *removed* any test files since you last ran `perl Makefile.PL` then you do not need to run that command again before running `make test`.

Conversely, whenever you do add or remove test files, you must remember to run `perl Makefile.PL` before you run `make test`.

`prove` always runs the files that are currently in `t/` at the time it is invokes.

Exercise

From within your `exercises/project/Client-Aggregate/` directory run your tests either using `prove` or `make test` as above.

Writing our own tests with Test::More

`Test::More` provides many great testing functions. We'll use some of these throughout this chapter, but you should read the documentation for more information.

```
use Test::More tests => 2;

# Test succeeds if first argument is true
ok( sum(1..10) == 55, "sum() works properly");

ok( product(1..10) == 3_628_880, "product() works properly");
```

When we run just this test we get:

```
ok 1 - sum() works properly
not ok 2 - product() works properly
#   Failed test ’product() works properly’
#   at ok.t line 10.
# Looks like you failed 1 test of 2.
```

As you can see, `ok` just determines whether its first argument is true. It can’t tell us where we’ve gone wrong. Thus, there is the `is` function:

```
use Test::More tests => 2;

# Test succeeds if first argument == second argument
is( sum(1..10), 55, "sum() works properly");

is( product(1..10), 3_628_880, "product() works properly");
```

When we run just this test we get:

```
1..2
ok 1 - sum() works properly
not ok 2 - product() works properly
#   Failed test ’product() works properly’
#   at ok.t line 10.
#          got: ’3628800’
#     expected: ’3628880’
# Looks like you failed 1 test of 2.
```

Now we get to compare. Here we can see that there’s an error in our test. We should have written:

```
is( product(1..10), 3_628_800, "product() works properly");
```

(that extra 8 can be hard to spot).

Having a plan

In the previous examples we told `Test::More` how many tests we were going to be running. We do this to guard against premature failure. If we tell `Test::More` how many tests we expect to run, then we set the plan up front:

```
use Test::More tests => 2;
```

If we don’t know how many tests we’re going to run we can just tell `Test::More` when we’ve finished testing:

```
use Test::More;

# tests

done_testing();
```

We can also set our plan once our test script is in action:

```
use Test::More;

# work out how many tests we want to run somehow...

plan tests => $number_of_tests;

# now do tests
```

or decide that we want to skip these tests based on some condition:

```
use Test::More;

if( $^0 eq "Win32" ) {
        plan skip_all => "Test not relevant for Windows 32 platform";
}
else {
        plan tests => 5;
}

# now do tests
```

When we `skip_all` tests, your script will output the reason why the tests are being skipped and then exit immediately with a zero (for success).

SKIP and TODO blocks

If instead of wanting to skip all the tests, we wish to only skip some, we can use a `SKIP` block:

```
SKIP: {
        skip $why, $how_many if $condition;
}
```

The condition is optional, but a good idea.

This might be useful if we're changing our code, and these tests might no longer apply, or if some of our tests depend on a module which hasn't been installed, or for all sorts of other reasons. The code inside a `SKIP` block will not be run at all.

We must specify how many tests we're skipping (`$howmany`) if we've set a plan so that the end count of tests ends up matching what `Test::More` expects.

If we have tests which fail, or which test unfinished code, we can use a `TODO` block. These are great because the tests are still run, but we expect them to fail so failures are okay.

```
TODO: {
        local $TODO = $why if $condition;

}
```

The condition is optional, and isn't often included.

Because the tests are still run, `Test::More` can flag tests when they succeed as unexpected successes. This means that you can write a bunch of tests for functionality that you don't yet have, and then as you add that functionality your tests will start being useful. Once you're happy with the results you can move the working tests outside the `TODO` block.

If you wish to see code using the `SKIP` and `TODO` blocks, look at both `exercises/project/Client-Aggregate/t/manifest.t` and `exercises/project/Client-Aggregate/t/t/boilerplate.t`.

Test data

Usually with tests, it's best to have a set of test data that won't change. We usually add this to a `data/` directory under our testing (`t/` directory. By having our test data here we can make certain assumptions in our tests about our data. For example we can be sure that we'll the same answer when looking up a given client id.

Exercise

For these exercises, read the documentation of both the `Client::Aggregate` and `Client::Record` classes. You can do this with `perldoc lib/Client::Aggregate` (and likewise for `Client::Record`) from the distribution level (`exercises/project/Client-Aggregate/`).

Having the documentation for `Test::More` open will also prove useful.

1. Make a `data/` directory in your testing directory (`t/`) and copy your `client-data.csv` file into it.
2. The following exercises will all be done in the same test file (which you put in the test directory). **We recommend you call it `01-subroutines.t`**

   To get you started, your file is going to need to contain:

   ```
   use strict;
   use warnings;

   use Test::More;
   use Client::Aggregate;
   use FindBin qw($Bin);
   ```

   You may choose to set a number of tests for `Test::More` (but you'll have to keep updating it), or alternately add `done_testing();` at the end of your program. Create this file.

   Run `perl Makefile.PL` in your distribution directory to add it to the Makefile. Now you can run `make test` or `prove -l t/` to run your tests.

   You can run just this test with `perl -Ilib t/01-subroutines.t`

   Run your tests after you complete each exercise, or more often.
3. Create a `Client::Aggregate` object by calling `read_file` and check that the result you get back is true (using `ok`). We recommend using `$Bin` from `FindBin` to specify the file location:

   ```
   my $clients = Client::Aggregate->read_file("$Bin/data/client_data.csv");
   ```
4. Write a test to check that the object type of the result we received above is `Client::Aggregate` (we can find out the type of an object with `ref` (eg `ref $clients`) so this should be a simple string comparison in an `is` test.
5. Use `where_id` to get the record for the first client. Check that it is true and a `Client::Record` object using `ref`. (Two tests).
6. Using the record you fetched in the previous question, write tests to show that the details match the known values (look at the actual data file and use the accessors on the `Client::Record` object). eg:

   ```
   is( $client->id(), 1, "ID matches" );
   ```

   You will need to write a test for each of the 12 accessors.
7. Ask for a client who doesn't exist (for example id = `5000`). Check that you get an undefined result.
8. 28 of our clients are based in NT. Use `where_state` to get these clients, and `how_many` to test that you have the right number.
9. Use `next` to get the first two clients from your NT search and check that their given names are: `Tyler` and `Caleb`. (Two tests).
10. Call `how_many` again on your NT result object. It should still be 28.

    This test will fail, and fixing this bug is **hard**. Put a `TODO` block around this test, and continue.

11. As it happens, we don't have any clients at all in Melbourne. Write a test to verify that we get an undefined value for this search.
12. In Spring Creek, Qld, we have 3 clients (2 male, 1 female). Fetch the male clients and verify you have only 2 values returned.
13. 4 of our clients come from Newcastle in NSW. Using `where_city` and `where_state` get these clients. Check that you get 4 clients.
14. Walk over each of your Novocastrians (people from Newcastle) by calling `next` and count how many iterations you make. Check that you get 4. (You'll need to keep a counter. We know that there are 4, but what if `next` somehow fails to give us the last one?).
15. `Test::More` can compare data structures for you with `is_deeply`. Use `is_deeply` to check that your Newcastle clients have the following ids `458`, `1584`, `1734`, `2166`. (Hint: look at the documentation).

Answers can be found in `exercises/answers/project/`.

Coverage testing and Devel::Cover

Ideally when developing your module, the more tests you have, the better. If you already have a test suite and you're wondering which parts of your code are not being tested, you can use the excellent `Devel::Cover` tool from http://search.cpan.org/perldoc?Devel::Cover.

When testing code coverage there are several different metrics that can be used. These include:

Statement coverage

> A statement is covered if it is executed. A statement is not necessarily the same as a line of code. A statement may also be a block of code or a segment. This is the weakest form of coverage.

Branch coverage

> A branch is covered if it is tested in with both true and false outcomes. A good testing suite will ensure that a program will jump to all possible destinations at least once.
>
> Branch coverage helps you spot missing else cases and cases where your least commonly executed blocks have invalid code in them. 100% branch coverage implies 100% statement coverage.

Path coverage

> Path coverage ensures that all paths through a program are taken. However this can lead to a great number of paths which are impossible to test. Often the path is restricted to paths in subroutines or consecutive branches.
>
> 100% path coverage implies 100% branch coverage, but this is hard.

Condition coverage

> Condition coverage requires testing all terms in each boolean expression. So `if($x || $y)` must be tested with each of the four possible values of `$x` and `$y`.
>
> 100% condition coverage implies 100% branch coverage.

Testing the code coverage of your test suite is bound to show up areas of your code that are never executed. Sometimes these areas will not be reachable. If this is the case then you've spotted an error in your implementation, design or specification. This is a good thing to spot.

`Devel::Cover` provides statement, branch, condition, subroutine, pod and time coverage. Using `Devel::Cover` on your test suite is easy. From the distribution directory we write:

```
cover -test
```

Running this tests will give us results like this:

```
--------------------------- ------ ------ ------ ------ ------ ------ ------
File                          stmt   bran   cond    sub    pod   time  total
--------------------------- ------ ------ ------ ------ ------ ------ ------
...read-multi/Text/CSV_XS.pm   23.4   13.7   10.3   21.3  100.0   32.5   23.2
blib/lib/Client/Aggregate.pm   85.2   77.8    n/a   75.0  100.0   35.9   84.1
blib/lib/Client/Record.pm     100.0    n/a    n/a  100.0  100.0   31.6  100.0
Total                          45.9   20.7   10.3   50.0  100.0  100.0   40.4
--------------------------- ------ ------ ------ ------ ------ ------ ------
```

and a HTML file in `Client-Aggregate/cover_db/coverage.html` This HTML file provides line by line coverage information and is well worth a perusal.

As you can see we haven't quite managed full coverage of our program. We can ignore `Devel::Cover`'s inclusion of `Text::CSV_XS`, but we can see that we've only covered 85% of our subroutines in `Client::Aggregate`. This suggests we need more tests. Covering all of our subroutines will probably improve our *stmt* and probably our *bran* coverage as well.

If you run `cover -t` and it says:

```
make: *** No rule to make target 'test'.  Stop.
```

and gives you no coverage information. Run `perl Makefile.PL` and try again.

`Devel::Cover` has some problems. Sometimes it'll attempt to test more modules than it should, it can fail with automatically generated methods and it can sometimes get upset about paths that include whitespace. It's great when it works, but is not perfect.

Exercise

Run `cover -test` on your code to determine your test coverage.

Good Tests

A good test is one which tests a single requirement from the specification. Such as: *the date field only accepts valid dates.* Obviously you'll probably need more than one test for this requirement.

The other kind of good test exists to test a single, known, restriction in the code. For example, the assumption that the name field won't have more than 100 characters in it. This may not be a requirement on your system but could be a consequence of your database design, web form or something else. Once again, you'll probably need more than one test for this restriction.

Tests that exist merely to achieve greater coverage are usually examples of bad tests.

A good test suite has one test for each aspect of each requirement and one test for each aspect of each known code restriction. In the examples above, this would include testing date-like data which can’t be correct (such as 31-31-2000) and valid dates. If there are restrictions on the minimum date or maximum date, either through specification and design or through implementation, there should be tests on the either side of the edge cases.

A well designed test suite will already have reasonable code coverage. Coverage testing then highlights the areas in which your test suite needs work. Code coverage can also highlight areas where your specification, design or implementation needs work.

Chapter summary

- Black box testing involves testing the requirements, not the implementation.
- White box testing involves testing the code with knowledge of its internals.
- `Test::More` provides a rich set of test functions with which we can test our code.
- `Devel::Cover` provides coverage testing capabilities for statement, condition, branch, subroutine and time coverage.

Chapter 15. External Files and Packages

In this chapter...

In this chapter we’ll discuss how we can split our code into separate files. We’ll discover Perl’s concept of packages, and how we can use them to make our code more robust and flexible.

Splitting code between files

When writing small, independent programs, the code can usually be contained within a single file. However there are two common occurrences where we would like to have our programs span multiple files. When working on a large project, often with many developers, it can be very convenient to split a program into smaller files, each with a more specialised purpose. Alternatively, we may find ourselves working on many programs that share some common code base. This code can be placed into a separate file which can be shared across programs. This saves us time and effort, and means that bug-fixes and improvements need to be made only in a single location.

Require

Perl implements a number of mechanisms for loading code from external files. The most simplest of these is by using the `require` function:

```
require ’file.pl’;
```

Perl is smart enough to make sure that the same file will not be included twice if it’s required through the same specified name.

```
# The file is only included once in the following case:
require ’file.pl’;
require ’file.pl’;
```

Required files *must* end with a true value. This is usually achieved by having the final statement of the file being:

```
1;
```

❗ Conflicts can occur if our included file declares subroutines with the same name as those that appear in our main program. In most circumstances the subroutine from the included file takes precedence, and a warning is given.

We will learn how to avoid these conflicts later in this chapter when we discuss the concept of *packages*.

The use of `require` has been largely deprecated by the introduction of modules and the `use` keyword. If you’re writing a code library from scratch we recommend that you create it as a module. However, `require` is often found in legacy code and is a useful thing to understand.

Any code in the file (except for subroutines) will be executed immediately when the file is required. The `require` occurs at run-time, this means that Perl will not throw an error due to a missing file until that statement is reached, and any subroutines inside the file will not be accessible until after the `require`.

Variables declared with `my` are not shared between files, they are only visible inside the block or file where the declaration occurs. To share packages between files we use *package variables* which are covered later in this chapter.

The use of modules (which we will learn about later) allows for external files to be loaded at compile-time, rather than run-time.

Use strict and warnings

Perl pragmas, such as `strict` and `warnings` are lexically scoped. Just like variables declared with `my`, they last until the end of the enclosing block, file or eval.

This means that you can turn strict and warnings on in one file without it influencing other parts of your program. Thus, if you’re dealing with legacy code, then your new libraries, modules and classes can be strict and warnings compliant even though the older code is not.

Example

The use of `require` is best shown by example. In the following we specify two files, `Greetings.pl` and `program.pl`. Both are valid Perl programs on their own, although in this case, `Greetings.pl` would just declare a variable and a subroutine, and then exit. As we do not intend to execute `Greetings.pl` on its own, it does not need to be made executable, or include a shebang line.

Our library code, to be included.

```
# Greetings.pl
# Provides the hello() subroutine, allowing for greetings
# in a variety of languages.  English is used as a default
# if no language is provided.

use strict;
use warnings;
use Carp;

my %greeting_in = (
        en      => "Hello",
        ’en-au’ => "G’day",
        fr      => "Bonjour",
        jp      => "Konnichiwa",
        zh      => "Nihao",
);
```

```
sub hello {
        my $language = shift || "en";

        my $greeting = $greeting_in{$language}
                or croak "Don’t know how to greet in $language";

        return $greeting;
}

1;
```

Our program code.

```
#!/usr/bin/perl
# program.pl
# Uses the Greetings.pl file to provide another hello() subroutine
use strict;
use  warnings;

# Get the contents from Greetings.pl
require "Greetings.pl";

print "English: ",    hello("en"),   "\n";       # Hello
print "Australian: ", hello("en-au"),"\n";       # G’day
```

Exercises

1. Create a file called `MyTest.pl` Define at least two subroutines; `pass` and `fail` which print some amusing output. Make sure that it uses `strict`.
2. Test that your code compiles by running **perl -c MyTest.pl**. (The **-c** tells Perl to check your code).
3. Create a simple Perl script (`caller.pl`)which requires `MyTest.pl` and calls the functions defined within.

Introduction to packages

The primary reason for breaking code into separate files is to improve maintainability. Smaller files are easier to work with, can be shared between multiple programs, and are suitable for dividing between members of large teams. However they also have their problems.

When working with a large project, the chances of naming conflicts increases. Two entirely different files may have two different subroutines with the same name; however it is only the last one loaded that will be used by Perl. Files from different projects may be re-used in new developments, and these may have considerable name clashes. Multiple files can also make it difficult to determine where subroutines are originally declared, which can make debugging difficult.

Perl’s *packages* are designed to overcome these problems. Rather than just putting code into separate files, code can be placed into independent packages, each with its own namespace. By ensuring that package names remain unique, we also ensure that all subroutines and variables can remain unique and easily identifiable.

A single file can contain multiple packages, but convention dictates that each file contains a package of the same name. This makes it easy to quickly locate the code in any given package.

Writing a package in Perl is easy. We simply use the `package` keyword to change our current package. Any code executed from that point until the end of the current file or block is done so in the context of the new package.

```
# By declaring that all our code is in the "Greetings" package,
# we can be certain not to step on anyone else’s toes, even if
# they have written a hello() subroutine.

package Greetings;

use strict;
use warnings;

my %greeting_in = (
        en      => "Hello",
        ’en-au’ => "G’day",
        fr      => "Bonjour",
        jp      => "Konnichiwa",
        zh      => "Nihao",
);

sub hello {
        my $language = shift || "en";

        my $greeting = $greeting_in{$language}
                or die "Don’t know how to greet in $language";

        return $greeting;
}

1;
```

The package that you’re in when the Perl interpreter starts (before you specify any package) is called `main`. Package declarations use the same rules as `my`, that is, it lasts until the end of the enclosing block, file, or eval.

Perl convention states that package names (or each part of a package name, if it contains many parts) starts with a capital letter. Packages starting with lower-case are reserved for pragmas (such as `strict`).

The scoping operator

Being able to use packages to improve the maintainability of our code is important, but there’s one important thing we have not yet covered. How do we use subroutines, variables, or filehandles from other packages?

Perl provides a *scoping operator* in the form of a pair of adjacent colons. The scoping operator allows us to refer to information inside other packages, and is usually pronounced "double-colon".

```
require "Greetings.pl";

# Greetings in English.
print Greetings::hello("en"),"\n";

# Greetings in Japanese.
print Greetings::hello("jp"),"\n";
```

```
# This calls the hello() subroutine in our main package
# (below), printing "Greetings Earthling".
print hello(),"\n";

sub hello {
        return "Greetings Earthling";
}
```

Calling subroutines like this is a perfectly acceptable alternative to exporting them into your own namespace (which we'll cover later). This makes it very clear where the called subroutine is located, and avoids any possibility of an existing subroutine clashing with that from another package.

Occasionally we may wish to change the value of a variable in another package. It should be very rare that we should need to do this, and it's not recommended you do so unless this is a documented feature of your package. However, in the case where we do need to do this, we use the scoping operator again.

```
use Carp;

# Turning on $Carp::Verbose makes carp() and croak() provide
# stack traces, making them identical to cluck() and confess().
# This is documented in 'perldoc Carp'.

$Carp::Verbose = 1;
```

There's a shorthand for accessing variables and subroutines in the `main` package, which is to use double-colon without a package name. This means that `$::foo` is the same as `$main::foo`.

When referring to a variable in another package, the sigil (punctuation denoting the variable type) always goes *before* the package name. Hence to get to the scalar `$bar` in the package `Foo`, we would write `$Foo::bar` and not `Foo::$bar`.

It is not possible to access lexically scoped variables (those created with `my`) in this way. Lexically scoped variables can *only* be accessed from their enclosing block.

Package variables and our

It is not possible to access lexically scoped variables (those created with `my`) outside of their enclosing block. This means that we need another way to create variables to make them globally accessible. These global variables are called *package variables*, and as their name suggests they live inside their current package. The preferred way to create package variables, under Perl 5.6.0 and above, is to declare them with the `our` statement. Of course, there are alternatives you can use with older version of Perl, which we also show here:

```
package Carp;

our $VERSION = '1.01';          # Preferred for Perl 5.6.0 and above

# use vars qw/$VERSION/;        # Preferred for older versions
# $VERSION = '1.01';

# $Carp::VERSION = '1.01';      # Acceptable but requires that we then
                                # always use this full name under strict
```

In all of the cases above, both our package and external code can access the variable using `$Carp::VERSION`.

Exercises

1. Change your `MyTest.pl` file to include a package name `MyTest`.
2. Update your calling program (`caller.pl`) to call the MyTest functions using the scoping operator (eg `pass()` now needs to be `MyTest::pass()`. Test that your calling code works.
3. Create a package variable `$PASS_MARK` using `our` inside `MyTest.pl` which defines an appropriate pass mark.
4. In your calling program (caller.pl), create a loop which tests 10 random numbers for pass or fail with reference to the `$PASS_MARK` package variable. Print the appropriate `pass` or `fail` message.
5. Print out the version of the `Cwd` module installed on your training server. The version number is in `$Cwd::VERSION`. (You will need to `use Cwd;` first.)
6. Look at the documentation for the `Carp` module using the **perldoc Carp** command. This is one of Perl's most frequently used modules.

Answers for the above exercises can be found in `exercises/answers/MyTest.pl` and `exercises/answers/packages.pl`.

You may receive the following warning during this exercise:

```
Name "MyTest::PASS_MARK" used only once: possible typo at ....
```

This occurs because perl does not load your external file until the `require` statement executes at run-time. However perl generates warnings about possible typos at compile time, before it loads your required file. We can avoid this by insisting perl load the external file at compile-time.

```
BEGIN { require ’MyTest.pl’; };
```

Modules (which we'll discuss next chapter) provide a cleaner and more extensible syntax for loading code at compile time.

Chapter summary

- A package is a separate namespace within Perl code.
- A file can have more than one package defined within it.
- The default package is `main`.
- We can get to subroutines and variables within packages by using the double-colon as a scoping operator for example `Foo::bar()` calls the `bar()` subroutine from the `Foo`
- To write a package, just write `package` *`package_name`* where you want the package to start.

- Package declarations last until the end of the enclosing block, file or eval (or until the next package statement).
- Package variables can be declared with the `our` keyword. This allows them to be accessed from inside other packages.
- The `require` keyword can be used to import the contents of other files for use in a program.
- Files which are included using `require` must end with a true value.

Chapter 16. Writing modules

In this chapter...

This chapter will discuss writing our own modules.

The rules for writing a module

You can read more about writing modules in **perldoc perlmodlib**, **perldoc perlmod**, **perldoc perlmodstyle**, and a little on pages 554-556 of the Camel book (3rd edition only).

As you've no doubt guessed by now, modules and packages often go hand-in-hand. We know how to use a module, but what are the rules on writing one? Well, the big one is this:

> A module is a file that contains a package of the same name.

So the `File::Copy` module contains a package called `File::Copy`. Our `MyTest` is almost a module.

Secondly, a module ends with the `.pm` extension, to tell Perl that it is a module.

Module naming

In the same way that we might organise documents into different folders for different projects; it is wise to organise modules into different name spaces. Thus you may have a module tree as follows:

```
ProjectA
    ::Finance
        ::CustomerAccounts
        ::Invoices
    ::Reports
        ::TimeWorked
        ::Income
```

Then each of these would contain packages of the appropriate name, and live on the file system in its own file. So we'd have the `ProjectA::Finance::CustomerAccounts` package inside `ProjectA/Finance/CustomerAccounts.pm`.

A module can contain multiple packages if you desire. So even though the module is called `Chess::Piece`, it might also contain packages for `Chess::Piece::Knight` and `Chess::Piece::Bishop`. It's usually preferable for each package to have its own module, otherwise it can be confusing to your users how they can load a particular package.

When writing modules, it's important to make sure that they are well-named, and even more importantly that they won't clash with any current or future modules, particularly those available via CPAN. If you are writing a module for internal use only, you can start its name with `Local::` which is reserved for the purpose of avoiding module name clashes.

The best way to get started with writing your own module is to use `Module::Starter`. Read its documentation on CPAN (http://search.cpan.org/perldoc?Module::Starter) and our mini-tutorial in our Perl Tips (http://perltraining.com.au/tips/2008-10-15.html). We'll also cover this in a later chapter.

To document your modules so that `perldoc` can provide information about them, read **perldoc perlpod** and **perldoc perlpodspec**.

Use versus require

Perl offers several different ways to include code from one file into another. `use` is built on top of `require` and has the following differences:

- Files which are `used` are loaded and executed at compile-time, not run-time. This means that all subroutines, variables, and other structures will exist before your main code executes. It also means that you will immediately know about any files that Perl could not load.
- `use` allows for the import of variables and subroutines from the used package into the current one. This can make programming easier and more concise.
- Files called with `use` can take arguments. These arguments can be used to request special features that may be provided by some modules.

Both methods:

- Check for redundant loading, and will skip already loaded files.
- Raise an exception on failure to find, compile or execute the file.
- Translate `::` into your system's directory separator.

Where possible `use` and Perl *modules* are preferred over `require`.

Warnings and strict

As the `warnings` and `strict` pragmas are lexically scoped, using them within your module will not affect your calling script. Likewise the use of `warnings` and `strict` in your calling script will not affect your module. As such you should always make sure your modules use `warnings` and `strict` explicitly.

```
use strict;
use warnings;
```

Exercise

This exercise will have you adapt your `MyTest.pl` code to become a module. There's a list at the end of this exercise of things to watch out for.

Making it into a module

1. Rename your `MyTest.pl` to be `MyTest.pm` (or copy it if you wish to keep the original).
2. Make sure `MyTest.pm` uses `strict` and `warnings`.
3. Test that your module has no syntax errors by running **perl -c MyTest.pm**.
4. Change your calling code (`caller.pl`) to `use` your new module, rather than requiring it. Check that everything works exactly as it should.

Moving your module into a subdirectory

1. Create a directory named `p5lib`.
2. Move your `MyTest.pm` file into your `p5lib` directory.
3. Add a `use lib` to your calling code (`caller.pl`) so that it can find your module. (`use lib 'p5lib';`)
4. Check that everything still works as you expect.
5. Add a print statement to your module (outside any subroutines). This should be printed when the module is loaded. Check that this is so.

Answers can be found in `exercises/answers/p5lib/MyTest.pm` and `exercises/answers/modules.pl`

Things to remember...

The above exercises can be completed without reference to the following list. However, if you're having problems, you may find your answer herein.

- A module is a file that contains a package of the same name.
- Perl modules must return a true value to indicate successful loading. (Put `1;` at the end of your module).
- To use a module stored in a different directory, add this directory to the `@INC` array. (Put `use lib 'path/to/modules/'` immediately before your `use MyTest;` line.
- To call a subroutine which is inside a module, you can access it via the double-colon. Eg: `MyModule::test();`

Exporting and importing subroutines

Writing your own import function for each and every module would be a tiresome and error-prone process. However, Perl comes with a module called Exporter, which provides a highly flexible interface with optimisations for the common case.

Exporter works by checking inside your module for three special data structures, which describe both what you wish to export, and how you wish to export them. These structures are:

- @EXPORT symbols to be exported into the user's name space by default
- @EXPORT_OK symbols which the user can request to be exported
- %EXPORT_TAGS that allows for bundles of symbols to be exported when the user requests a special export tag.

@ISA

To take advantage of Exporter's import function we need to let Perl know that our package has a special relationship with the Exporter package. We do this by telling Perl that we *inherit* from Exporter. Our package and the rest of our program does not need to be written in an object oriented style for this to work.

Now when Perl goes looking for the import function it will first look in our package. If it can't be found there, Perl will look for a special array called @ISA. The contents of the @ISA array is interpreted as a list of parent classes, and each of these will be searched for the missing method.

To specify that this package is a sub-class of the Exporter module we include the following lines:

```
use Exporter;
our @ISA = qw(Exporter);
```

use base

An alternative to adding parent modules to @ISA yourself is to use the base pragma. This allows you to declare a derived class based upon the listed parent classes. Thus the two lines above becomes:

```
use base qw(Exporter);
```

The base pragma takes care of ensuring that the Exporter module is loaded.

The base pragma is available for all versions of Perl above 5.6.0.

An example

Here's an example of just using @EXPORT and @EXPORT_OK. Our hypothetical module, People::Manage is used for managing interpersonal relations.

```
package People::Manage;
use base qw(Exporter);

our @EXPORT     = qw(invite $name @friends %addresses);        # invite is a subroutine
our @EXPORT_OK  = qw(&taunt $spouse @enemies %postcodes);      # so is taunt
```

The ampersand in front of subroutines is optional.

Exporting by default

Exporting your symbols by default, by populating the `@EXPORT` array, means that anyone using your module will receive these symbols without having to ask for them. This is generally considered to be bad style, and is sometimes referred to as ’polluting’ the caller’s namespace.

The reason this is considered to be bad style is that there is nothing in the `use` line to indicate that anything is being exported. A programmer who is not familiar with the module may inadvertently define their own subroutines or variables which clash with those that are exported. Likewise, a reviewer examining the code will not easily be able to determine from which module a given subroutine may have been exported, especially if many modules are used.

Using the `@EXPORT` array is highly discouraged.

Using `@EXPORT_OK` allows the user to choose which symbols they wish to bring into their name space. All other symbols can be accessed by using their full name, such as `People::Manage::invite()`, when required.

An example

Our module:

```
###### People/Manage.pm ###########
package People::Manage;         # create a package of the same name
use strict;
use warnings;
use base qw(Exporter);

# List out the things we wish to export
our @EXPORT_OK  = qw(invite $name @friends %addressbook
                     taunt $spouse @children $pet);

# Only package variables can be exported, as such all of these
# variables need to be declared with ’our’ not ’my’.

our $name     = "Fred";
our $spouse   = "Wilma";
our @children = qw(Pebbles);
our @friends  = qw(Barney Betty);
our $pet      = "Dino";
my $address   = "301 CobbleStone Way, Bedrock";

our %addressbook = (
        Barney         => "303 Cobblestone Way, Bedrock",
        Betty          => "303 Cobblestone Way, Bedrock",
        "Barney’s Mom" => "142 Boulder Ave, Granitetown",
);

sub invite {
        my ($friend, $date) = @_;
        return "Dear $friend,\n $spouse and I would love you to come to".
               "dinner at our place ($address) on $date.\n\n".
               "Yours sincerely, $name\n";
}

sub taunt {
        my ($enemy) = @_;
        return "Dear $enemy, my pet $pet has more brains than you.\n";
}

1;                              # module MUST end with something true
```

Our program:

```
###### dinner.pl ###########
#!/usr/bin/perl
use strict;
use warnings;
use People::Manage qw(invite %addressbook);     # only using a few things

# Invite some people over for dinner.

foreach my $person (keys %addressbook) {
        print invite($person,"next Tuesday");
}
```

Importing symbols

Once your module is written and it exports a few symbols, it's time to use it. This is done with the `use` command that we've seen with `strict` and other modules. We can load our module in three ways:

- `use People::Manage;` which imports all of the symbols stored in `@People::Manage::EXPORT`.
- `use People::Manage ();` which imports *none* of the symbols in either `@People::Manage::EXPORT` or `@People::Manage::EXPORT_OK`.
- `use People::Manage qw($name $spouse invite);` which imports all the listed symbols. If a symbol is mentioned which is not in either `@People::Manage::EXPORT` or `@People::Manage::EXPORT_OK` then a fatal error will occur.

Exercises

These exercises build on the previous exercises.

1. Change your MyTest.pm module to export the `pass` and `fail` symbols and import those into your script. Change your script to call `pass` and `fail` instead of their fully qualified names.
2. Change your module to export the `$PASS_MARK` variable and use that instead of its fully qualified name.

Exporting tags

If you wish to export groups of symbols that are related to each other, there is an `%EXPORT_TAGS` hash which provides this functionality. This can be used in the following manner:

```
%EXPORT_TAGS = (family => [ qw/$name $spouse @children $pet/ ],
                social => [ qw/%addressbook invite taunt @friends/ ],
);
```

Names which appear in `%EXPORT_TAGS` must also appear in `@EXPORT` or `@EXPORT_OK`. Tags themselves cannot be used in either export array.

Importing symbols through tags

Symbols grouped in tags can be imported normally, by specifying each symbol, or by using the tag provided. This is done by prepending the tag name with a colon:

```
use People::Manage qw(:family);          # Family related information.
use People::Manage qw(:social);          # Social-related symbols.
use People::Manage qw(:family :social);  # Both
```

Writing your own documentation

A module is not complete without the documentation. If you used `Module::Starter` to generate your starting files, much of the scaffolding will have been written for you. In either case, it's good to have an understanding of how it all works.

All of Perl's documentation is written in a format called Plain Old Documentation (POD). POD is designed to be a simple, clean documentation language. It uses three kinds of instructions.

- Text paragraphs which the formatter can reformat if required. This is the default form of documentation.
- Code paragraphs which the formatter must not reformat. These are signalled by starting each line in the paragraph with at least one white space character.
- Command paragraphs. These must start in the first column, be preceeded and followed by a blank line and start with an equals character (=).

```
package My::Module;
use strict;

# Generally your code goes up here. Your POD may be at the bottom
# (as in this example), but you can interleave your POD and your code
# if you so wish.  Interleaving POD and code can make it easier to keep
# your subroutine documentation up-to-date.

# Any command paragraph (which starts with an equals) causes Perl to
# consider all lines to be POD until it sees the =cut directive.

=head1 NAME

My::Module - Show an example of Perl documentation.

=head1 SYNOPSIS

    use My::Module qw(example pirate);

    example($foo, $bar);
    pirate($yarr, $avast);

=head1 DESCRIPTION

This documentation is an example.  It shows different headings,
code paragraphs, and even lists!  Here’s a list now:
```

```
=over 4

=item *

Bullet lists are easy.

=item *

See L<perlpod> to see how to do numbered lists and definition lists.

=back

=head1 SUBROUTINES

=head2 example

    my $baz = example($foo, $bar);

Each function should have its own heading and a brief description of what
it does, its arguments and what it returns.

=head2 pirate

    my $booty = pirate($yarr, $avast);

It be a fine day for writing subroutine documentation ’pon the high seas.

=head1 AUTHOR

Perl Training Australia

=cut

1;
```

We can then render the POD part of this file, using **perldoc** just as we did to read Perl’s normal documentation. **perldoc My/Module.pm**:

```
My::Module(1)        User Contributed Perl Documentation       My::Module(1)


NAME
       My::Module - Show an example of Perl documentation.

SYNOPSIS
           use My::Module qw(example pirate);

           example($foo, $bar);
           pirate($yarr, $avast);

DESCRIPTION
       This documentation is an example.  It shows different headings, code
       paragraphs, and even lists!  Here’s a list now:

       o   Bullet lists are easy.

       o   See perlpod to see how to do numbered lists and definition lists.

SUBROUTINES
       example

           my $baz = example($foo, $bar);

       Each function should have its own heading and a brief description of
       what it does, its arguments and what it returns.

       pirate

           my $booty = pirate($yarr, $avast);

       It be a fine day for writing subroutine documentation ’pon the high
       seas.

AUTHOR
       Perl Training Australia


perl v5.8.8                       2009-03-19                     My::Module(1)
```

To make sure your POD is valid, you can use the **podchecker** which comes standard with perl.

To learn more about POD command paragraphs read **perldoc perlpod** and our Perl Tip (http://perltraining.com.au/tips/2005-04-06.html).

Exercise

Document your `MyTest` module. Make sure you add the NAME, SYNOPSIS, DESCRIPTION, SUBROUTINES and AUTHOR fields.

Chapter summary

- Modules can be used for class definitions or as libraries for common code.
- A module can contain multiple packages, but this is often a bad idea.
- It’s often a good idea to put your own modules into the `Local` namespace or a project specific namespace.

Chapter 17. Building modules with Module::Starter

In this chapter...

Although it is possible to create a module from scratch directly, as we did in the previous chapter, there is much more work to be done. A good module should have a build system, documentation, a test suite, and numerous other bits and pieces to assist in its easy packaging and development. These are useful even if we never release our module to CPAN.

Setting this up can be a lot of work, especially if you’ve never done it before. Yet, we want to spend our time writing code not trying to get the set-up perfect. That’s where `Module::Starter` comes in handy. It provides a simple, command-line tool to create a skeleton module quickly and easily.

Using module-starter

Before we can build our module with `module-starter` we need to install `Module::Starter` from the CPAN. `Module::Starter` allows us to choose from a variety of build frameworks, from the aging `ExtUtils::MakeMaker` to `Module::Install` and `Module::Build`. While `ExtUtils::MakeMaker` comes standard with Perl, you may need to install the other build frameworks, although `Module::Build` comes standard with Perl 5.10.0 and above. For this chapter we will use `Module::Install`.

Creating a module with `Module::Starter` couldn’t be easier. On the command line we simply write:

```
    module-starter --module=My::Module --author="Jane Smith" \
--email=jane.smith@example.com --builder=Module::Install
```

The module name, author, and e-mail switches are all required. We’ve used the optional `--builder` switch to specify we want to use `Module::Install` as our build-system, instead of the default `ExtUtils::MakeMaker`.

Once this is done, you should have a `My-Module` directory with a skeleton module inside.

Configuration

`module-starter` will look for a configuration file first by checking the `MODULE_STARTER_DIR` environment variable (to find a directory containing a file called `config`), then by looking in `$HOME/.module-starter/config`. This can be used to provide the required arguments to make module creating even easier.

The configuration file is just a list of names and values. Lists are space separated. For example we might have:

```
author:  Jane Smith
email:   jane.smith@example.com
builder: Module::Install
```

With this configuration file, the above call to `module-starter` becomes:

```
module-starter --module=My::Module
```

Exercise

For this chapter, we are going to make a module that doesn't do much. Run `module-starter` with appropriate values and call your module `My::Module`.

A skeleton tour

If you've never created a module before, or you've been making them by hand, then it's nice to take a look at what you get for your `Module::Starter` skeleton.

```
My-Module$ ls -la

total 36
drwxr-xr-x  4 jarich jarich 4096 2010-03-16 17:01 .
drwxrwxrwt 15 jarich jarich 4096 2010-03-16 17:01 ..
-rw-r--r--  1 jarich jarich  109 2010-03-16 17:01 Changes
-rw-r--r--  1 jarich jarich   96 2010-03-16 17:01 .cvsignore
drwxr-xr-x  3 jarich jarich 4096 2010-03-16 17:01 lib
-rw-r--r--  1 jarich jarich  200 2010-03-16 17:01 Makefile.PL
-rw-r--r--  1 jarich jarich   90 2010-03-16 17:01 MANIFEST
-rw-r--r--  1 jarich jarich 1379 2010-03-16 17:01 README
drwxr-xr-x  2 jarich jarich 4096 2010-03-16 17:01 t

Let's look at each of these files in turn:
```

Changes

> This is a human-readable file tracking module revisions and changes. If you're going to release your code to the CPAN, it's essential for your users to know what has changed in each release. Even if you're only using your code internally, this is a good place to document the history of your project.

.cvsignore

> `Module::Starter` assumes you'll be using CVS for revision control, and provides a `.cvsignore` file with the names of files that are auto-generated and not to be tracked with revision control. If you use git for new projects, and you could rename this to `.gitignore`.

lib/

> The `lib/` directory will contain your skeleton module, and is where you'll be doing much of your work. `Module::Starter` will have already added some skeleton documentation, a version number, and some skeleton functions.
>
> You can add more modules to the `lib/` directory if you wish. Splitting a very large module into smaller, logical pieces can significantly improve maintainability.

MANIFEST

> The `MANIFEST` file tracks all the files that should be packaged when you run a `make tardist` to distribute your module. Normally it includes your source code, any file needed for the build system, a `META.yml` that contains module meta-data (usually auto-generated by your build system), tests, documentation, and anything else that you want your end-user to have.
>
> If you don't want to manually worry about adding entries to the `MANIFEST` file yourself, most build systems (including `Module::Install`) allow you to write `make manifest` to auto-generate

it. For this to work, you'll want to make a `MANIFEST.skip` file which contains filenames and regular expressions that match files which should be excluded from the `MANIFEST`.

Makefile.PL

This is the front-end onto our build system. When we wish to build, test, or install our module, we'll always invoke `Makefile.PL` first:

```
perl Makefile.PL
make
make test
make install
```

Most build systems will provide a `make tardist` target for building a tarball of all the files in our `MANIFEST`, a `make disttest` for making sure our tests work with only the `MANIFEST` listed files, and `make clean` and `make distclean` targets for clearing up auto-generated files, including those from the build system itself if a `make distclean` is run.

You'll almost certainly wish to customise your `Makefile.PL` a little, especially if your module has dependencies. You'll want to consult your build system documentation for what options you can uses. For `Module::Install` this documentation can be found at http://search.cpan.org/perldoc?Module::Install.

README

The `README` file should contain basic information for someone thinking of installing your module. Mentioning dependencies, how to build, and how to find/report bugs are all good things to mention in the `README` file. Some systems (including the CPAN) will extract the `README` and make it available separate from the main distribution.

t/

The `t/` directory contains all the tests that will be executed when you run a `make test`. By default, `Module::Starter` will provide some simple tests to ensure that your module compiles, that you'll filled in relevant sections of the boilerplate documentation, and that your documentation covers all your subroutines and doesn't contain any syntax errors.

Local Installation

At this stage we have a fully working, but completely non-functional module. We can install this locally using the `--INSTALL_BASE` option when we invoke `Makefile.PL`.

To start with, we may need to create the directories we want to install our modules into. For example (in our home directory, or preferred directory):

```
mkdir perl5
mkdir perl5/lib
```

Then we can install our module as covered earlier:

```
perl Makefile.PL INSTALL_BASE=perl5
make
make test
make install
```

Exercise

Create the above directories, and install your module into those local directories. (If you are using Windows you may need to use `dmake` instead of `make`).

Further information

If you’re thinking of releasing your code to the CPAN, or just want more information about preparing a module for distribution, you can learn more by reading the `perlnewmod` page with `perldoc perlnewmod` or on-line at http://search.cpan.org/perldoc?perlnewmod.

Chapter Summary

- `module-starter` reduces much of the excess work in creating a module, by providing scaffolding and template files.
- A configuration file can be created to ensure that all modules are created with the same initial information.
- Modules can be installed locally by passing the `INSTALL_BASE` flag to `Makefile.PL`.
- When creating large systems, it pays to carefully consider your intended module interface before writing any code.

Chapter 18. References and complex data structures

In this chapter...

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

Assumed knowledge

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

Introduction to references

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

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

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

- Chapter 8 (chapter 4, 2nd Ed) of the Camel book, and in **perldoc perlref**.
- Chapter 1 of Advanced Perl Programming (O'Reilly's Panther book).

Uses for references

There are three main uses for Perl references.

Creating complex data structures

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

Passing arrays and hashes to subroutines and functions

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

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

printargs(@a1, @a2);

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

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

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

Object oriented Perl

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

Creating and dereferencing references

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

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

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

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

```
print $scalar_ref;               # prints SCALAR(0x80c697c)
print $array_ref;                # prints ARRAY(0x80c6988)
print $hash_ref;                 # prints HASH(0x80c69e2)
```

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

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

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

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

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

In other words, wherever you would normally put a variable name (like `@array`) you can put a reference variable (like `@$array_ref`). For example:

```
# Example of adding items to an array via a reference.
push(@$colours_ref, ’cyan’, ’magenta’);

# Example of getting a list of keys via a hash reference.
my @friends = keys %$friends_ref;
```

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

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

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

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

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

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

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

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

Exercises

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

Assigning through references

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

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

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

my %fruit = (
        kumquat  => ’sour’,
        orange   => ’sweet’,
        lemon    => ’sour’,
        mandarin => ’sweet’,
);
my $fruit_ref = \%fruit;

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

Passing multiple arrays/hashes as arguments

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

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

two_lists(@chosen, @colours);

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

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

        ...

        return;
}
```

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

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

sub ref_two_lists {
        my ($chosen_ref, $colours_ref) = @_;

        print "Chosen list:\n";
        foreach (@$chosen_ref) {
                print "$_\n";
        }

        print "Colour list:\n";
        foreach (@$colours_ref) {
                print "$_\n";
        }
        return;
}
```

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

```
sub ref_two_lists {
        my ($chosen_ref, $colours_ref) = @_;

        my @chosen_copy  = @$chosen_ref;         # this @chosen is now a copy
        my @colours_copy = @$colours_ref;        # this @colours is now a copy
        ...

        return;
}
```

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

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

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

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

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

Anonymous data structures

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

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

# if we get rid of $array_ref, @array will still hang round using up
# memory.  Here’s how we do it without the intermediate step, by
# creating an anonymous array:
my $array_ref = [’a’, ’b’, ’c’, ’d’];

# look, we can still use qw() too...
my $array_ref = [ qw(a b c d) ];

# Print the letter ’b’ via the reference.
print $array_ref->[1], "\n";

# more useful yet, we can put these anon arrays straight into a hash:
my %transport = (
        cars          =>   [qw(toyota ford holden porsche)],
        planes        =>   [qw(boeing harrier)],
        boats         =>   [qw(clipper skiff dinghy)],
);

# Print "skiff" via our transport hash.
print $transport{boats}->[1],"\n";
```

The same technique can be used to create anonymous hashes:

```
# The old, two-step way:
my %hash = (
        a       =>   1,
        b       =>   2,
        c       =>   3,
);
my $hash_ref = \%hash;

# the quicker way, with an anonymous hash:
my $hash_ref = {
        a       =>   1,
        b       =>   2,
        c       =>   3,
};
```

Data is pulled out of an anonymous data structure using the arrow notation:

```
print $hash_ref->{a};       # prints "1";
```

Exercise

1. Change your previous program to initialise `$friends_ref` using an anonymous array constructor. You should no longer need your original `@friends` array. Test that your program still works.

Complex data structures

You can find more about complex data structures in Appendix B and also by reading both **perldoc perldsc** and **perldoc perllol**.

References are most often used to create complex data structures. Since references are scalars, they can be used as values in both hashes and arrays. This makes it possible to create both deep and complex multi-dimensional data structures. These are covered more deeply in Appendix C.

The use of references in data structures allows you to create arrays of arrays, arrays of hashes, hashes of arrays and hashes of hashes. We saw an example of a hash of arrays in the previous section. Here is an example of an array of hashes:

```
my %alice = (
        name           => "Alice Jane",
        age            => 34,
        employeenumber => 12003,
);
my %bob = (
        name           => "Bob Jane",
        age            => 32,
        employeenumber => 12345,
);


my @employees = (
        \%alice,
        \%bob,
);

# to print out Alice’s employee number:
print $employees[0]->{employeenumber};

# Or, to use anonymous data structures
my @employees2 = (
        {
                name           => "Alice Jane",
                age            => 34,
                employeenumber => 12003,
        },
        {
                name           => "Bob Jane",
                age            => 32,
                employeenumber => 12345,
        },
);

# to print out Bob’s age:
print $employees2[1]->{age};
```

Exercises

There is a starter file for these exercises in `exercises/food_starter.pl`. You may also find it useful to read Appendix B.

1. Create data structures as follows:
   a. Create a hash called `%pasta_prices` which contains prices for small, medium and large serves of pasta.
   b. Create a hash called `%milkshake_prices` which contains prices for small, medium and large milkshakes.
   c. Edit the `%menu` hash to contain references to the above hashes, so that given a type of food and a size you can find the price of it. Don’t forget that your hash must contain both keys (the type of food), and values (a reference to the data structure containing the prices).
2. Print out the cost of a large pizza by referencing your `%menu` hash.
3. Code already exists to accept the food-type and size from the user. Change the print line so that it prints the correct price for that food choice.
4. Convert the menu hash to use anonymous data structures (a hash of hashes) instead of the original three pizza, pasta and milkshake hashes. Check that your customer code works with this change.
5. Add a new element to your menu hash which contains the prices of salads. Rather than adding this in when you create the hash, instead add it separately.
6. Create a subroutine which can be passed a scalar and a hash reference. Check whether there is an element in the hash which has the scalar as its key. Hint: use `exists` for this.

Answers for the above exercises can be found in `exercises/answers/food.pl`.

Dereferencing expressions

We have already seen how to turn a reference back into the original variable using the syntax `$$scalar_ref`, `@$array_ref` and `%$hash_ref`. These work fine for individual variables, but what if we have a subroutine that returns an array reference, and we want back the original array?

Perl allows us to use the syntax `${...}`, `@{...}` and `%{...}` to take an *expression* and dereference the result into a scalar, array, or hash respectively:

```
my $friends_ref = get_friends_arrayref();

foreach my $friend (@$friends_ref) {
        print "Hi $friend!\n";
}

# The same, but using @{ ... }

foreach my $friend ( @{get_friends_arrayref()} ) {
        print "Hi $friend!\n";
}
```

While you may see this syntax in other people’s programs, the resulting code is often more complex and difficult to understand. Any time you consider dereferencing an expression, consider whether your code would be easier to read if you assigned to a separate variable and deferenced that, as in the first example above.

Copying complex data structures

In general, copying a Perl variable is easy:

```
my $scalar_copy = $scalar;
my @array_copy  = @array;
my %hash_copy   = %hash;

my $array_ref_copy = $array_ref;
```

However this only copies at the top level which for arrays and hashes means copying the array elements, or key-value pairs. If the array elements or hash values are references, then a copy to the memory location the reference represents is made, rather than a copy of that underlying structure. This means that if we modify the resultant copy, we may modify the original structure. To avoid this we can use `Storable`'s `dclone` function.

```
use strict;
use Storable qw(dclone);

my %transport = (
        cars          =>   [qw(toyota ford holden porsche)],
        planes        =>   [qw(boeing harrier)],
        boats         =>   [qw(clipper skiff dinghy)],
);

# Take a deep copy of our hash (dclone returns a reference)
my $transport_copy = dclone \%transport;

# Add a type of boat to our copied hash
$transport_copy->{boats}[3] = "sloop";

# We can also add using push
push @{ $transport_copy->{boats} }, "raft";

# Print boats from transport_copy hash:
my @new_boats = @{ $transport_copy->{boats} };
print "@new_boats\n";      # prints "clipper skiff dinghy sloop raft"

# Print boats from original transport hash:
my @original_boats = @{ $transport{boats} };
print "@original_boats\n"; # prints "clipper skiff dinghy"
```

Data::Dumper

Typically, to print out a data structure you have to understand its underlying structure and then write a number of loops to print it out in full. If the structure is relatively simple such as a hash of hashes of values, or even a hash of hash of arrays this isn't too difficult.

However, often data structures are very complex, and negotiating and printing these structures can be a tiresome exercise. It's also an unnecessary one, as all the hard work has already been done for you. To save you from having to write specialised printing code in every program for debugging purposes, there's a special library you may find useful called `Data::Dumper`.

`Data::Dumper` provides a function which takes Perl data structures and turns them into human readable strings representing the data with in them. It can be used just like this:

```
use Data::Dumper;

my %HoH = (
        Jacinta => {
                age               => 26,
                favourite_colour => "blue",
                sport             => "swimming",
                language          => "Perl",
        },
        Paul => {
                age               => 27,
                favourite_colour => "green",
                sport             => "cycling",
                language          => "Perl",
        },
);
print Dumper \%HoH;
```

This will print out something similar to:

```
$VAR1 = {
          'Paul' => {
                      'language' => 'Perl',
                      'favourite_colour' => 'green',
                      'sport' => 'cycling',
                      'age' => 27
                    },
          'Jacinta' => {
                         'language' => 'Perl',
                         'favourite_colour' => 'blue',
                         'sport' => 'swimming',
                         'age' => 26
                       }
        };
```

Not only is this easy to read, but it's also perfectly valid Perl code. This means you can use `Data::Dumper` to easily give you a structure that you can paste into another program, or which can be 'serialised' to a file and re-created at a later date. `Data::Dumper` has a lot more uses beyond simple debugging.

`Dumper` expects to be given one or more references to data structures to dump. If `Dumper` is provided with a hash or array then every element of the array, or every key and value of the hash, will be considered a separate data structure, and dump separately. The results are not particularly useful:

```
# result of: print Dumper %HoH;
$VAR1 = 'Paul';
$VAR2 = {
          'language' => 'Perl',
          'favourite_colour' => 'green',
          'sport' => 'cycling',
          'age' => 27
        };
$VAR3 = 'Jacinta';
$VAR4 = {
          'language' => 'Perl',
          'favourite_colour' => 'blue',
          'sport' => 'swimming',
          'age' => 26
        };
```

You can read more about `Data::Dumper` on page 882 of the Camel book or in **perldoc Data::Dumper**.

Exercises

The transport hash from the previous example can be found in `exercises/references.pl`. Use this file for these exercises.

1. Use `Data::Dumper` to print out the `%transport` and `%models` data structures.
2. Print out the second plane in the `%transport` hash.
3. Print out all of the planes in the `%transport` hash.
4. Print out the third boeing model from the `%models` hash.
5. Print out all of the holden car models in the `%models` hash.
6. Print out all of forms of transport from the `%models` hash.
7. Print out all of the boat manufacturers from the `%models` hash.
8. Use **perldoc Data::Dumper** to read about `Data::Dumper`'s many options and configuration variables.

Chapter summary

- References are scalar data consisting of the memory address of a piece of Perl data, and can be used in arrays, hashes, and other places where you would use a normal scalar
- References can be used to create complex data structures, to pass multiple arrays or hashes to subroutines, and in object-oriented Perl.
- References are created by prepending a backslash to a variable name.
- References are dereferenced by replacing the name part of a variable name (eg `foo` in `$foo`) with a reference, for example replace `foo` with `$foo_ref` to get `$$foo_ref`
- References to arrays and hashes can also be dereferenced using the arrow `->` notation.
- References can be passed to subroutines as if they were scalars.
- References can be included in arrays or hashes as if they were scalars.
- Anonymous arrays can be made by using square brackets instead of round; anonymous hashes can be made by using curly brackets instead of round. These can be assigned directly to a reference, without any intermediate step.
- `Data::Dumper` allows complex data structures to be printed out verbatim without requiring full knowledge of the underlying data structure.

Chapter 19. Advanced regular expressions

In this chapter...

This chapter builds on the basic regular expressions taught earlier in the course. We will learn how to handle data which consists of multiple lines of text, including how to input data as multiple lines and different ways of performing matches against that data.

Assumed knowledge

You should already be familiar with the following topics:

- Regular expression meta characters
- Quantifiers
- Character classes and alternation
- The `m//` matching function
- The `s///` substitution function
- Matching strings other than `$_` with the `=~` matching operator

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

Capturing matched strings to scalars

Perl provides an easy way to extract matched sections of a regular expression for later use. Any part of a regular expression that is enclosed in parentheses is captured and stored into special variables. The substring that matches first set of parentheses will be stored in `$1`, and the substring that matches the second set of parentheses will be stored in `$2` and so on. There is no limit on the number of parentheses and associated numbered variables that you can use.

```
/(\w)(\w)/;     # matches 2 word characters and stores them in $1, $2
/(\w+)/;        # matches one or more word characters and stores them in $1
```

Parentheses are numbered from left to right by the *opening* parenthesis. The following example should help make this clear:

```
$_ = "fish";
/((\w)(\w))/;   # captures as follows:
                # $1 = "fi", $2 = "f", $3 = "i"

$_ = "1234567890";
/(\d)+/;        # matches each digit and then stores the last digit
                # matched into $1
/(\d+)/;        # captures all of 1234567890
```

Evaluating a regular expression in list context is another way to capture information, with parenthesised sub-expressions being returned as a list. We can use this instead of numbered variables if we like:

```
$_ = "Our server is training.perltraining.com.au.";
my ($full, $host, $domain) = /(([\w-]+)\.([\w.-]+))/;
print "$1\n";                  # prints "training.perltraining.com.au."
print "$full\n";               # prints "training.perltraining.com.au."
print "$2 : $3\n";             # prints "training : perltraining.com.au."
print "$host : $domain\n"      # prints "training : perltraining.com.au."
```

In earlier versions of Perl, there were efforts to attempt to set the match variables ($1, $2 etc) to undefined if a match failed. Unfortunately this was difficult and not always successful. In more recent versions of Perl, this attempt has been reversed, and so failed matches *will not* reset the match variables.

You should *always* check that your regular expression succeeded before using any data from it.

Consider the following (wrong) code:

```
while(<>) {
        # check that we have something that looks like a date in
        # YYYY-MM-DD format.

        if(/(\d{4})-(\d{2})-(\d{2})/) {
                print STDERR "valid date\n";
        }
        next unless $1;

        if($1 >= $recent_year) {
                print RECENT_DATA $_;
        }
        else {
                print OLD_DATA $_;
        }
}
```

If this code encounters a line which doesn't appear to be a valid date, the line may be printed to the same file as the last valid line, rather than being discarded. This could result in lines with dates similar to "1901-3-23" being printed to `RECENT_DATA`, or lines with dates like "2003-1-1" being printed to `OLD_DATA`.

The above would be better written as:

```
while(<>) {
        # check that we have something that looks like a date in
        # YYYY-MM-DD format.

        if(/(\d{4})-(\d{2})-(\d{2})/) {
                if($1 >= $recent_year) {
                        print RECENT_DATA $_;
                }
                else {
                        print OLD_DATA $_;
                }
        }
        else {
                print STDERR "Unexpected line: $_\n";
        }
}
```

Extended regular expressions

Regular expressions can be difficult to follow at times, especially if they're long or complex. Luckily, Perl gives us a way to split a regular expression across multiple lines, and to embed comments into our regular expression. These are known as *extended regular expressions*.

To create an extended regular expression, we use the special `/x` switch. This has the following effects on the match part of an expression:

- Spaces (including tabs and newlines) in the regular expression are ignored.
- Anything after an un-escaped hash (`#`) is ignored, up until the end of line.

Extended regular expressions do not alter the format of the second part in a substition. This must still be written exactly as you wish it to appear.

If you need to include a literal space or hash in an extended expression you can do so by preceeding it with a backslash.

By using extended regular expressions, we can change this:

```
# Parse a line from 'ls -l'
m{^([\w-]+)\s+(\d+)\s+(\w+)\s+(\w+)\s+(\d+)\s+(\w+\s+\d+\s+[\d:]+)\s+(.*)$};
```

into this:

```
# Parse a line from 'ls -l'

m{
    ^                           # Start of line.
    ([\w-]+)\s+                 # $1 - File permissions.
    (\d+)\s+                    # $2 - Hard links.
    (\w+)\s+                    # $3 - User
    (\w+)\s+                    # $4 - Group
    (\d+)\s+                    # $5 - File size
    (\w+\s+\d+\s+[\d:]+)\s+     # $6 - Date and time.
    (.*)                        # $7 - Filename.
    $                           # End of line.
}x;
```

As you can see, extended regular expressions can make your code much easier to read, understand, and maintain.

Exercise

Web server access logs typically contain long lines of information, only some of which is of interest at any given time. In the `exercises/access-pta.log` file you'll see an example taken from Perl Training Australia's webserver.

1. Write a regular expression which captures the request origin, the access date and requested page. Print this out for each access in the file. A starting program can be found in `exercises/log-process.pl`.

You can find an answer to this exercise in `exercises/answers/log-process.pl`.

Advanced exercise

1. Split tab-separated data into an array then print out each element using a `foreach` loop (an answer's in `exercises/answers/tab-sep.pl`, an example file is in `exercises/tab-sep.txt`).

Greediness

Regular expressions are, by default, "greedy". This means that any regular expression, for instance `.*`, will try to match the biggest thing it possibly can. Greediness is sometimes referred to as "maximal matching".

Greediness is also left to right. Each section in the regular expression will be as greedy as it can while still allowing the whole regular expression to match if possible. For example,

```
$_ = "The cat sat on the mat";

/(c.*t)(.*)(m.*t)/;

print $1;               # prints "cat sat on t"
print $2;               # prints "he "
print $3;               # prints "mat";
```

It is possible in this example for another set of matches to occur. The first expression `c.*t` could have matched `cat` leaving ` sat on the ` to be matched by the second expression `.*`. However, to do that, we need to stop `c.*t` from being so greedy.

To make a regular expression quantifier not greedy, follow it with a question mark. For example `.*?`. This is sometimes referred to as "minimal matching".

```
$_ = "The fox is in the box.";

/(f.*x)/;              # greedy               -- $1 = "fox is in the box"
/(f.*?x)/;             # not greedy           -- $1 = "fox"

$_ = "abracadabra";

/(a.*a)/               # greedy               -- $1 = "abracadabra"
/(a.*?a)/              # not greedy           -- $1 = "abra"

/(a.*?a)(.*a)/         # first is not greedy  -- $1 = "abra"
                       # second is greedy     -- $2 = "cadabra"

/(a.*a)(.*?a)/         # first is greedy      -- $1 = "abracada"
                       # second is not greedy -- $2 = "bra"

/(a.*?a)(.*?a)/        # first is not greedy  -- $1 = "abra"
                       # second is not greedy -- $2 = "ca"
```

Exercise

1. Write a regular expression that matches the first and last words on a line, and print these out. A file of test data can be found in `exercises/regextest.txt`.

More meta characters

Here are some more advanced meta characters, which build on the ones covered earlier.

Table 19-1. More meta characters

Meta character	Meaning
`\c`*`x`*	Control character, i.e. **CTRL-*x***
`\0`*`nn`*	Octal character represented by *nn*
`\x`*`nn`*	Hexadecimal character represented by *nn*
`\l`	Lowercase next character
`\u`	Uppercase next character
`\L`	Lowercase until `\E`
`\U`	Uppercase until `\E`
`\Q`	Quote (disable) meta characters until `\E`
`\E`	End of lowercase/uppercase/quote
`\A`	Beginning of string, regardless of whether /m is used.
`\Z`	End of string (or before newline at end), regardless of whether /m is used.
`\z`	Absolute end of string, regardless of whether /m is used.

```
# search for the C++ computer language:

/C++/         # wrong! regexp engine complains about the plus signs
/C\+\+/       # this works
/\QC++\E/     # this works too

# search for "bell" control characters, eg CTRL-G

/\cG/         # this is one way
/\007/        # this is another -- CTRL-G is octal 07
/\x07/        # here it is as a hex code
```

Read about all of these and more in **perldoc perlre**.

Working with multi-line strings

Often, you will want to read a file several lines at a time. Consider, for example, a typical Unix fortune cookie file, which is used to generate quotes for the **fortune** command:

```
All language designers are arrogant.  Goes with the territory... :-)
             -- Larry Wall in <1991Jul13.010945.19157@netlabs.com>
%
Although the Perl Slogan is There’s More Than One Way to Do It, I hesitate
to make 10 ways to do something.  :-)
             -- Larry Wall in <9695@jpl-devvax.JPL.NASA.GOV>
%
And don’t tell me there isn’t one bit of difference between null and space,
because that's exactly how much difference there is.  :-)
             -- Larry Wall in <10209@jpl-devvax.JPL.NASA.GOV>
%
"And I don't like doing silly things (except on purpose)."
             -- Larry Wall in <1992Jul3.191825.14435@netlabs.com>
%
:        And it goes against the grain of building small tools.
Innocent, Your Honor.  Perl users build small tools all day long.
             -- Larry Wall in <1992Aug26.184221.29627@netlabs.com>
%
/* And you'll never guess what the dog had */
/*   in its mouth... */
             -- Larry Wall in stab.c from the perl source code
```

The fortune cookies are separated by a line which contains nothing but a percent sign.

To read this file one item at a time, we would need to set the delimiter to something other than the usual `\n` - in this case, we'd need to set it to something like `\n%\n`.

To do this in Perl, we use the special variable `$/`. This is called the input record separator.

```
$/ = "\n%\n";
while (<>) {
        # $_ now contains one RECORD per loop iteration
}
```

Conveniently enough, setting `$/` to `""` will cause input to occur in "paragraph mode", in which two or more consecutive newlines will be treated as the delimiter. Undefining `$/` will cause the entire file to be slurped in.

```
undef $/;
$_ = <>; # whole file now here
```

Changing `$/` doesn't just change how readline (`<>`) works. It also affects the `chomp` function, which always removes the value of `$/` from the end of its argument. The reason we normally think of `chomp` removing newlines is that `$/` is set to newline by default.

It's usually a very good idea to use `local` when changing special variables. For example, we could write:

```
{
        local $/ = "\n%\n";
        $_ = <>;        # first fortune cookie is in $_ now
}
```

to grab the first fortune cookie. By enclosing the code in a block and using local, we restrict the change of `$/` to that block. After the block `$/` is whatever it was before the block (without us having to save it and remember to change it back). This localisation occurs regardless of how you exit the block, and so is particularly useful if you need to alter a special variable for a complex section of code.

Variables changed with `local` are also changed for any functions or subroutines you might call while the `local` is in effect. Unless it was your intention to change a special variable for one or more of the subroutines you call, you should end your block before calling them.

It is a compile-time error to try and declare a special variable using `my`.

Special variables are covered in Chapter 28 of the Camel book, (pages 127 onwards, 2nd Ed). The information can also be found in **perldoc perlvar**.

Since `$/` isn't the easiest name to remember, we can use a longer name by using the **English** module:

```
use English;

$INPUT_RECORD_SEPARATOR = "\n%\n";      # long name for $/
$RS = "\n%\n";                          # same thing, awk-like
```

The **English** module is documented on page 884 (page 403, 2nd Ed) of the Camel book or in **perldoc English**. You can find out about all of Perl's special variables' English names by reading **perldoc perlvar**.

Exercise

1. In your directory is a file called `exercises/perl.txt` which is a set of Perl-related fortunes, formatted as in the above example. This file contains a great many quotes, including the ones in the example above and many many more. Use a regular expression to find only those quotes which are from the `perl man page`. (Answer: `exercises/answers/fortunes.pl`)

Regexp modifiers for multi-line data

Perl has two modifiers for multi-line data. `/s` and `/m`. These can be used to treat the string you're matching against as either a single line or as multiple lines. Their presence changes the behaviour of caret (`^`), dollar (`$`) and dot (`.`).

By default caret matches the start of the string. Dollar matches the end of the string (regardless of newlines). Dot matches anything but a newline character.

With the `/s` modifier, caret and dollar behave the same as in the default case, but dot will match the newline character.

With the `/m` modifier, caret matches the start of any line within the string, dollar matches the end of any line within the string. Dot does not match the newline character.

```
my $string = "This is some text
and some more text
spanning several lines";

if ($string =~ /^and some/m) {                  # this will match because
        print "Matched in multi-line mode\n";   # ^ matches the start of any
}                                               # line in the string

if ($string =~ /^and some/) {                   # this won't match
        print "Matched in single line mode\n";  # because ^ only matches
}                                               # the start of the string.

if($string =~ /^This is some/) {                # this will match
        print "Matched in single line mode\n";  # (and would have without
}                                               # the /s, or with /m)
```

```
if($string =~ /(some.*text)/s) {   # Prints "some text\nand some more text"
        print "$1\n";              # Note that . is matching \n here
}

if($string =~ /(some.*text)/)   {   # Prints "some text"
        print "$1\n";               # Note that . does not match \n
}
```

The differences between default, single line, and multi-line mode are set out very succinctly by Jeffrey Friedl in Mastering Regular Expressions (see the Further Reading at the back of these notes for details). The following table is paraphrased from the one on page 236 of that book.

His term "clean multi-line mode" describes one in which each of `^`, `$` and `.` all do what many programmers expect them to do. That is `.` will match newlines as well as all other characters, and `^` and `$` each work on start and end of lines, rather than the start and end of the string.

Table 19-2. Effects of single and multi-line options

Mode	Specified with	`^` matches...	`$` matches...	Dot matches newline
default	neither `/s` nor `/m`	start of string	end of string	No
single-line	`/s`	start of string	end of string	Yes
multi-line	`/m`	start of line	end of line	No
clean multi-line	both `/m` and `/s`	start of line	end of line	Yes

Modifiers may be clumped at the end of a regular expression. To perform a search using “clean multi-line” irrespective of case your expression might look like this

```
/^the start.*end$/msi
```

and if we had the following strings

```
$string1 = "the start of the day
is the end of the night";

$string2 = "10 athletes waited,
the starting point was ready
how it would end
was anyone’s guess";

$string3 = uc($string2); # same as string 2 but all in uppercase
```

we’d expect the match to succeed with both `$string2` and `$string3` but not with `$string1`.

Regular expression special variables

PREMATCH, MATCH and POSTMATCH

There are a few special variables related to regular expressions. The parenthesised names beside them are their long names if you use the English module.

- `$&` is the matched text (MATCH)

- `$‘` (dollar backtick) is the unmatched text to the left of the matched text (PREMATCH)
- `$’` (dollar forwardtick) is the unmatched text to the right of the matched text (POSTMATCH)

These variables are modified when a match occurs, and can be used in the same way that other scalar variables can be used.

```
my ($match) = m/^(\d+)/;
print $match;

# the same thing, using $&;
m/^\d+/;
print $&;
```

Efficiency concerns

When Perl sees you using PREMATCH (`$‘`), MATCH (`$&`), or POSTMATCH (`$’`), it assumes that you may want to use them again. This means that it has to prepare these variables after every successful pattern match, for every regular expression in your program. This can slow a program down because these variables are "prepared" by copying the string you matched against to an internal location.

If the use of those variables make your life much easier, then go ahead and use them. However, if using `$1`, `$2` etc can be used for your task instead, your program will be faster and leaner by using them. For example:

```
my ($prematch, $match, $postmatch) = /(.*?) (\d{20}) (.*)/xs;

print "before: $prematch\n";
print "match: $match\n";
print "after: $postmatch\n";

# prints the same as below, without penalising other expressions

/\d{20}/;

print "before: $‘\n";
print "match: $&\n";
print "after: $’\n";
```

The Devel::SawAmpersand (http://search.cpan.org/perldoc?Devel::SawAmpersand) and Devel::FindAmpersand (http://search.cpan.org/perldoc?Devel::FindAmpersand) modules can be used to determine if and where any of the MATCH/PREMATCH/POSTMATCH variables were used.

Perl 5.10+ alternative

If you are using Perl 5.10 or above, you can use the special variables `${^PREMATCH}`, `${^MATCH}` `${^POSTMATCH}` instead, for any regular expression with the `/p` modifier. These are *only* set for regular expressions with the `/p` modifier, and thus do not have the same performance penalties as `$‘`, `$&` and `$’`.

```
/(foo|bar|baz)/p;

say "Everything before the match: ${^PREMATCH}";
say "Everything inside the match: ${^MATCH}";
say "Everything after  the match: ${^POSTMATCH}";
```

Capture variables

We’ve already seen that if we use parentheses inside regular expressions that the match inside those parentheses is stored in the special match variables `$1`, `$2`, `$3`... There are as many match variables as we need.

```
# match the first three words...
m/^(\w+) (\w+) (\w+)/;
print "$1 $2 $3\n";
```

If you want to use parentheses simply for grouping, and don’t want them to set a `$1` style variable, you can use a special kind of *non-capturing* parentheses, which look like `(?: ... )`

```
# this only sets $1 - the first set of parentheses are non-capturing
m/(?:Dr|Prof) (\w+)/;
```

You can also use `$1` and other special variables in substitutions:

```
$string = "It was a dark and stormy night.";
$string =~ s/(dark|wet|cold)/very $1/;

# swap first and second words
s/^(\w+) (\w+)/$2 $1/;
```

Back-references

While we can use our capture variables in substitutions, this is no use in a simple match pattern, because `$1` and friends aren’t set until after the match is complete. Something like:

```
print if m{(t\w+) $1};
```

will *not* match "this this" or "that that". Rather, it will match a string containing "this" followed by whatever `$1` was set to by an earlier match.

In order to match "this this" (or "that that") we need to use the special regular expression meta characters `\1`, `\2`, etc. These meta characters refer to parenthesised parts of a match pattern, just as `$1` does, but *within the same match* rather than referring back to the previous match.

```
# print if we find repeated words starting with ’t’, eg: "this this"
# (note, this contains a subtle bug which you’ll find in the exercise)

print if m{(t\w+) \1};
```

Exercises

1. Write a script which swaps the first and the last words on each line. You may find the test file in `exercises/regextest.txt` helpful.

2. Write a script which looks for doubled terms such as "bang bang" or "quack quack" and prints out all occurrences. This script could be used for finding typographic errors in text. (Answer: `exercises/answers/double.pl`)

Advanced exercises

1. Make your swapping-words program work with lines that start and end with punctuation characters. (Answer: `exercises/answers/firstlast.pl`)
2. Modify your repeated word script to work across line boundaries (Answer: `exercises/answers/multiline_double.pl`)
3. What about case sensitivity with repeated words?

Chapter summary

- Input data can be split into multi-line strings using the special variable `$/` , also known as `$INPUT_RECORD_SEPARATOR`.
- The `/s` and `/m` modifiers can be used to treat multi-line data as if it were a single line or multiple lines, respectively. This affects the matching of `^` and `$` , as well as whether or not `.` will match a newline.
- The special variables `$&`, `$‘` and `$’` are always set when a successful match occurs.
- `$1`, `$2`, `$3` etc are set after a successful match to the text matched by the first, second, third, etc sets of parentheses in the regular expression. These should only be used *outside* the regular expression itself, as they will not be set until the match has been successful.
- Special non-capturing parentheses `(?:...)` can be used for grouping when you don’t wish to set one of the numbered special variables.
- Special meta characters such as `\1`, `\2` etc may be used *within* the regular expression itself, to refer to text previously matched.

Chapter 20. File I/O

In this chapter...

In this chapter, we learn how to open and interact with files.

Angle brackets

The line input operator

The line input operator is discussed in-depth on page 81 (page 53, 2nd Ed) of the Camel book. You can read about the closely-related `readline` function using **perldoc -f readline**.

We have encountered the line input operator <> in situations such as these:

```
# reading lines from STDIN (or from files on the command line)
while (<>) {
        # Process the line of input in $_
}

# reading a single line of user input from STDIN
my $input = <STDIN>;

# reading all lines from STDIN into an array
my @input = <STDIN>;
```

- In scalar context, the line input operator yields the next line of the file referenced by the filehandle given.
- In list context, the line input operator yields all remaining lines of the file referenced by the filehandle. (Be careful when using this as you may use up all your memory if the file is large).
- The default filehandle is `STDIN`, or any files listed on the command line of the Perl script (eg **myscript.pl file1 file2 file3**).

Opening a file for reading, writing or appending

The `open()` function is documented on pages 747-755 (pages 191-195, 2nd Ed) of the Camel book, and also in **perldoc -f open**.

The `open()` function is used to open a file for reading or writing (amongst other things).

In brief, Perl uses the same characters as shell does for file operations. That is:

- < says to open the file for reading
- > says to open the file for writing
- >> says to open the file for appending.

If you need more control over how you open your files, check out the `sysopen` function by using **perldoc -f sysopen**. Using `sysopen` is especially important if you’re running with elevated privileges, as it can help protect against dangerous race conditions. You can read more about this on pages 571-573 in the Camel book (3rd Ed only).

Opening and reading from a file

In a typical situation, we might use `open()` to open a file for reading:

```
# Open our file for reading

my $logfile_fh;

open($logfile_fh, ’<’, ’/var/log/httpd/access.log’)
        or die "Failed to open logfile for reading: $!";
```

The first argument we pass to `open` is empty scalar. Perl creates a *filehandle* and places it into this scalar for us. The filehandle is used to access the file after it has been opened.

In our example, since the file has been opened for reading, we can now use the readline operator (<>) with that filehandle, just as we did when we read from `STDIN`. It is also possible to declare our scalar and pass it to `open` in a single step:

```
# Open our file for reading
open(my $logfile_fh, ’<’, ’/var/log/httpd/access.log’)
        or die "Failed to open logfile for reading: $!";

# Now read from our file
while(<$logfile_fh>) {
        # do something with $_
}
```

Instead of reading into `$_` we can instead read into a variable name of our choice too:

```
# Open our file for reading
open(my $logfile_fh, ’<’, ’/var/log/httpd/access.log’)
        or die "Failed to open logfile for reading: $!";

# Now read each line into our own variable
while(my $line = <$logfile_fh>) {
        # do something with $line
}
```

Opening files for writing and appending

We use > and >> to open files for writing:

```
# Open file for writing
open(my $out_fh, '>', '/tmp/output') or die $!;

# Open file for appending
open(my $append_fh, '>>', '/tmp/out.log') or die $!;
```

When using > to open files for writing this will *clobber* any contents of your file. > truncates the file when it is opened, just as it does in the shell. So even if you don't write anything to the file, the original contents will be lost upon opening.

Using > or >> will cause the files to spring into existence if they do not already exist, so you don't have to worry about how to create them before writing.

Once we have a file open for either writing or appending we can write to them using `print` by adding a special type of argument called an *indirect argument*.

```
# Open file for writing
open(my $out_fh, '>', '/tmp/output') or die $!;

# Write to file
print $out_fh "This will be printed to the file.\n";
```

Note that there is no comma between our filehandle and the string we want to print. This is because our filehandle is an indirect argument to `print`. If there were a comma, we would instead print both our filehandle and our string to STDOUT instead.

It is both possible and encouraged to place the indirect argument in a block to make it stand out further, and reduce the temptation to add a comma in by accident:

```
# Open file for writing
open(my $out_fh, '>', '/tmp/output') or die $!;

# Write to file (with a block to make the filehandle stand out)
print {$out_fh} "This will be printed to the file.\n";
```

Closing your file

To close a file you have finished working with, use the `close` function:

```
close($out_fh);
```

Closing a file that has been opened for writing or appending makes sure that any buffered data has been written to that file.

Perl will automatically close scalar filehandles that fall out of scope.

It is possible that Perl may be unable to close your file, for example if your file is on a network file server and your network connection has failed. In such instances `close` will return a false value to signal error. You can test for this as follows:

```
close($out_fh) or die "Failed to close output file: $!";
```

Failure

You should *always* check for failure of an `open()` statement:

```
open(my $logfile_fh, ’<’, ’/var/log/httpd/access.log’)
        or die "Can’t open /var/log/httpd/access.log: $!";
```

Attempting to read from or write to an unopened file may cause unexpected results and is almost certainly not what you want your code to be doing.

`die` is a Perl function which takes an error message and terminates the program displaying that message to the user. In this example, the die statement is executed only if the open statement returns false; in other words, if there was an error in opening the file. `$!` is the special variable which contains the error message produced by the last system interaction.

Perl tries to be helpful when dying on errors and will append the appropriate filename and line number of your script to the end of the die message, with a newline. If you don’t want this behaviour, end the die message with a newline (`\n`) character. For example:

```
# The following provides an error with file and line-number:
open(my $logfile_fh, "<", $file) or die "Cannot open $file: $!";

# Here the file and line-number are omitted.
open(my $loglife_fh, "<", $file) or die "Cannot open $file: $!\n";
```

Make sure you don’t do this by accident, and miss out on this important information.

`$!` is documented on page 669 (page 134, 2nd Ed) of the Camel book and also in **perldoc perlvar**.

You can read more about `die` on page 700 (page 157, 2nd Ed) of the Camel book and also with **perldoc -f die**.

Using autodie to handle failure

An alternative to explicitly checking whether `open` and other functions succeeded is to use the `autodie` module. `autodie` causes Perl’s built-in functions to throw an exception upon failure (die) instead of just returning false. If the exception is not caught, your program will die with an human-readable error similar to what you would have written yourself. Like `strict` and `warnings`, the `autodie` pragma lasts until the end of the current block or file.

```
use autodie;

open(my $logfile_fh, ’<’, $file);   # no need to check!
close($logfile_fh);                 # no need to check!

{
        no autodie;     # Turn off autodie just for this block.

        # Without autodie, we do our own error handling.
        open(my $logfile_fh, ’<’, $file)
                or die "Failed open $file for reading: $!";
}
```

The `autodie` pragma is a standard module in Perl 5.10.1 and and can be installed from the CPAN for the 5.8.x or 5.10.0 releases of Perl.

For more information read **perldoc autodie** or the CPAN documentation (http://search.cpan.org/perldoc?autodie).

Instead of `autodie`, the `Fatal` module can be used instead. It comes standard with all versions of Perl and behaves similarly. However care must be taken when using `Fatal` on existing programs. It changes the behaviour of the specified functions for the whole package, not necessity just the part you’re looking at.

For more information read **perldoc Fatal**.

Exercises

1. Write a script which opens a file for reading. Use a `while` loop to print out each line of the file.
2. Use the above script to open a Perl script. Use a regular expression to print out only those lines not beginning with a hash character (i.e. non-comment lines). (Answer: `exercises/answers/delcomments.pl`)
3. Create a new script which opens a file for writing. Write out the numbers 1 to 100 into this file. (Hint: the numbers 1 to 100 can be generated by using the `..` operator eg: `foreach my $value (1..100) {}`) (Answer: `exercises/answers/100count.pl`)
4. Create a new script which opens a log file for appending. Create a `while` loop which accepts input from STDIN and appends each line of input to the log file. (Answer: `exercises/answers/logfile.pl`)
5. Create a script which opens two files, reads input from the first, and writes it out to the second. (Answer: `exercises/answers/readwrite.pl`)

Package filehandles (legacy)

In legacy code you may find package filehandles being used where we’ve used scalar filehandles above. These have no sigil out the front and are written in all capital letters:

```
use autodie;

# Open access.log for reading using LOGFILE as our filehandle
open(LOGFILE, ’<’, ’/var/log/httpd/access.log’);

# use the filehandle in the <> line input operator to read the
# contents
while (<LOGFILE>) {
        print if m{ \Q perltraining.com.au \E }x;
}

close LOGFILE;
```

Package filehandles have a package-wide scope rather than the lexical (per block or per file) scope in our earlier examples. This means that if you attempt to open to a filehandle of the same name as a previously opened file, then you will close the previous file; possibly causing subtle bugs. Consider the following example:

```
use autodie;

# Open access.log for reading using LOGFILE as our filehandle
open(LOGFILE, '<', '/var/log/httpd/access.log');

# use the filehandle in the <> line input operator to read the
# contents
while (my $line = <LOGFILE>) {
        print $line if test_line($line);
}

close LOGFILE;

#...
# many lines later
#...

sub test_line {
        my ($line) = @_;

        if( $line =~ m{ \Q perltraining.com.au \E }x) {
                # Trying to debug an error...
                open LOGFILE, ">>", "/tmp/log");
                print LOGFILE $line;

                return $line;
        }
        return;
}
```

The addition of the debugging code to log which lines we're matching is going to cause our program to only be able to match one line ever. Once we have a successful match we close the original file from earlier in our program, in order to open a file with the same filehandle for writing. We may write to that file, but our next *read* in the while loop fails and the loop terminates. If we've turned on warnings, we will be told that we're attempting to read from a filehandle open for writing, otherwise this code may appear to work some of the time, but fail at other times.

Package filehandles are also very difficult to pass as arguments to subroutines or store in hashes and arrays. Their use is strongly discouraged for any new code.

Scalar filehandles before Perl 5.6.0

Scalar filehandles, as we have used above, are a built-in feature for all versions of Perl from 5.6.0 and above. If your version of perl is older than that, you can still use scalar filehandles, by using the `FileHandle` module, but you need to declare your intentions first:

```
use FileHandle;

my $fh = FileHandle->new;     # $fh is now a FileHandle object.
open ($fh, "<", "/path/to/file") or die $!;

my $out_fh = FileHandle->new;
open ($out_fh, ">", "/path/to/other/file") or die $!;
```

The `FileHandle` module comes standard with Perl; and gives you no excuse for writing code with block-style filehandles.

Using the `FileHandle` module also works in Perl 5.6.0 and above, so if compatibility with older versions of Perl is important to you, you should use the `FileHandle` module for scalar filehandles.

For more information see `perldoc FileHandle` and pages 895-898 (page 442-444, 2nd Ed) in the Camel book.

Two argument open

In the examples above, we've used three-argument `open`. With three-argument `open`, the filename is treated literally, so if you try to `open` a file called `fred; echo Hello world` then Perl will look for a file with that exact name (which probably won't exist).

Three-argument `open` only exists in Perl from version 5.6.0 and above.

You may also see two-argument `open` in Perl code. Two argument `open` combines the mode and the filename as the second argument and ignores all leading and trailing whitespace.

```
open(my $logfile_fh, "< /var/log/httpd/access.log")
        or die "Can't open /var/log/httpd/access.log: $!";
```

If we open a file without specifying the mode (by using <, > or >>) Perl assumes we want to read from the file. Thus you will occasionally see:

```
# DON'T DO THIS!
open(my $logfile_fh, '/var/log/httpd/access.log')
        or die "Can't open /var/log/httpd/access.log: $!";
```

In this particular case, it is safe for Perl to assume you want to read from the file. Despite this, it is **always** a good idea to explicitly open your files for reading by using the < character; especially if you are accepting the filename from your user, as the following code is a security issue:

```
# REALLY DON'T DO THIS!
my $filename = <STDIN>;
open(my $logfile_fh, $filename) or die "Can't open $filename: $!";
```

Doing this allows the user to set how the file is opened. If the file starts with > or >> then the file will be opened for writing or appending respectively. If the file starts or ends with `|` the user can run code on your system! It is better to always specify the mode yourself!

For a safe file open for those who can't upgrade to Perl 5.6, have a look at `sysopen`. Information about sysopen can be found in **perldoc -f sysopen** and pages 808-810 (page 194, 2nd Ed) of the Camel book.

Changing file contents

When manipulating files, we may wish to change their contents. A flexible way of reading and writing a file is to import the file into an array, manipulate the array, then output each element again.

It is important to ensure that should anything go wrong we don't lose our original data. As a result, it's considered *best-practice* to write our data out to a temporary file and them move that over the input file after everything has been successful.

```
# a program that reads in a file and writes the lines in sorted order.
use autodie;

open(my $infile, "<", "file.txt");
my @lines = <$infile>;   # Slurps all the lines into @lines.
close $infile;

@lines = sort @lines;

# open temporary file to save our sorted data into
open(my $outfile, ">", "file.txt.tmp");

# use print's ability to print lists
print {$outfile} @lines;
close $outfile;

# we know that we were successful, so write over the original file
# only move the file *after* the filehandle has been closed.
rename("file.txt.tmp", "file.txt");
```

You can learn more about Perl's `rename` function with **perldoc -f rename**.

Always remember to close the file before attempting to `rename`. Failure to do this may result in `rename` attempting to move or copy the file before all of the data has been written to it, or for the rename to fail entirely on systems that don't allow open files to be renamed.

If you need to work with whole files which may be very big, the above method can consume a lot of memory. For such cases, it's worth looking at `Tie::File` which represents a regular text file as a Perl array without pre-loading the whole file into memory. Read **perldoc Tie::File** for more information.

Secure temporary files

The `File::Temp` module creates a name and filehandle for a temporary file. The default assumption is that any such temporary file will be a binary file. In this example we'll be using Perl's `binmode` function to mark it as a text file when needed. We'll discuss more about `binmode` later in this chapter.

```
use File::Temp qw(tempfile);

my ($tmp_fh, $tmp_name) = tempfile();

# Set the file as a text-file on Win32 systems.
binmode($tmp_fh,':crlf') if ($^O eq "MSWin32");

print {$tmp_fh} @lines;
close $tmp_fh;

# only move the file *after* the filehandle has been closed.
rename($tmp_name, "file.txt");
```

The `File::Temp` module can also be used to create in-memory temporary files if required.

Looping over file contents

If you don't need to manipulate all of the lines together (for example sorting) you ought to forgo the reading things into an array and just loop over each line. Continue to ensure, however, that your original data cannot be lost if the program terminates unexpectedly.

```
use autodie;

# removes duplicated lines
open(my $infile, "<", "file.txt");
open(my $outfile, ">", "unique.txt");

my $prevline = "";
while(<$infile>) {
        print {$outfile} $_ unless ($_ eq $prevline);
        $prevline = $_;
}

close $infile;
close $outfile;
```

Exercises

1. The `exercises/numbers.txt` contains a single number on each line. Open the file for reading, read in each line and increment the number by 1. Print the results to a second file.
2. Now that the above program is working, use `rename` to save your changes back to the original file name. Make sure you are closing your filehandle before moving the file! (Answer: `exercises/answers/increment.pl`)
3. Open a file, reverse its contents (line by line) and write it back to the same filename. For example, "this is a line" would be written as "enil a si siht" (Answer: `exercises/answers/reversefile.pl`)

Opening files for simultaneous read/write

Files can be opened for simultaneous read/write by putting a + in front of the > or < sign. +< is almost always preferable, as +> would clobber the file before you had a chance to read from it.

Read/write access to a file is not as useful as it sounds --- except under special circumstances (notably when dealing with fixed-length records) you can’t usefully write into the middle of the file using this method, only onto the end. The main use for read/write access is to read the contents of a file and then append lines to the end of it.

```
# Example: Reading a file and adding to the end.
# Program that checks to see if $username appears in the file
# adds $username to the end, if not.
use autodie;

my $username = <STDIN>;
chomp $username;

open(my $users_fh, "+<", "users.txt");

my $found;
while(<$users_fh>) {
        chomp;

        # case insensitive matching
        if(lc($_) eq lc($username)) {
                $found = 1;
                last;
        }
}

# We’ll be at the end of our file if $found isn’t set
unless($found) {
        print {$users_fh} "$username\n";
}
close $users_fh;
```

The small print

+< puts you at the start of the file. Note that it won’t create a new file if the file you’re dealing with does not exist (you’ll just get an error that the file doesn’t exist). If you start writing before you’ve reached the end of the file, you will overwrite characters in that file (from that point). Even if you’re dealing with fixed-length records and think you know what you’re doing, this is often still a bad idea.

+>> initially puts you at the end of the file. It will create a new file if necessary and will not clobber an old one. It allows you to read at any point in the file, but all writes will always go to the end.

Buffering

When Perl wants to read a file from disk, it asks the operating system to go fetch it. It would be very slow if Perl had to ask the operating system for each and every line, so typically it asks for a large chunk (a block) and then holds it in memory, until your program has used it all and needs more, or has finished executing. This is called input buffering.

For the same reasons, Perl also buffers its output. That is, it saves up the data that you want to print to a file or STDOUT and only prints it when it has enough. Once the buffer is full, an end of file character is seen or the filehandle is closed, then it is *flushed* to the disk. This is why it is essential that we close the relevant filehandle before renaming a file. Data going to the screen rather than a file will be sent upon seeing a newline.

You can see the effects of buffering with the following code (in `exercises/buffering.pl`):

```
foreach my $number (1..5) {
        print "$number ";
        sleep(1);
}
```

STDERR, on the other hand, is never buffered. When you print to STDERR your content appears on the screen or in the file immediately. We can also turn off buffering to our other filehandles when necessary. To do so, we can use Perl's `IO::Handle` module.

```
use IO::Handle;

# Turn on automatic flushing for STDOUT
STDOUT->autoflush(1);

# Flush $some_filehandle’s buffer, but don’t turn on autoflush
$some_filehandle->flush();

# Turn on automatic flushing for $fh
$fh->autoflush(1);

# Turn off automatic flushing STDOUT (now it’ll be buffered again)
STDOUT->autoflush(0);
```

Since both print and readline (<>) are buffered, you should *not* use them for editing a file in-place. If you must work with in-place edits, use the lower level functions such as `sysseek()`, `syswrite()` and `sysread()`. Perl also has a `-i` switch, for more useful in-place modification of files. These concepts are not covered in this course.

For more information about `open` including simultaneous read/write, see **perldoc perlopentut**. Also read pages 747-755 (pages 191-195, 2nd Ed) of the Camel book.

For information about the `-i` option to Perl read **perldoc perlrun** and pages 495-497 (page 332, 2nd Ed) of the Camel book.

Read the documentation in **perldoc IO::Handle** for the standard way in Perl to control buffering on a per-filehandle basis.

An excellent tutorial on buffering, its advantages and disadvantages, and how to manipulate it from Perl can be found in Mark Jason Dominus' excellent article on *Suffering from Buffering* available from his Perl FAQs (http://perl.plover.com/FAQs/Buffering.html).

Opening pipes

If the filename given to two-argument `open()` begins with a pipe symbol (`|`), the filename is interpreted as a command to which output is to be piped, and if the filename ends with a `|`, the filename is to be interpreted as a filename which pipes input to us.

We can use pipes to read information from any process we can execute on our system. Once the command is open, we can read from the resulting filehandle in the same way we would read from any other file. In the example below, we use secure shell (**ssh**) to read a file on a remote machine.

```
#!/usr/bin/perl
# This program allows us to read a file from another machine
# using secure shell.  This is most useful if we can login without
# a password (eg, established keys).
use strict;
use warnings;
use autodie;

# Process our command line arguments, and complain if we don’t
# have both a host and filename.  Note that we’re explicitly trusting our
# users to behave themselves with their inputs.
my ($host, $file) = @ARGV;
unless ($host and $file) {
        die "Usage: $0 host filename\n";
}
open (my $ssh_fh, "ssh $host cat $file |");

while(<$ssh_fh>) {
        # We can process the file in any way we like here.
        # In this particular case, we’ll simply print it to
        # our STDOUT.

        print;
}
```

Here’s an example which writes to the **sort** command, which is a standard utility on both Windows and Unix systems. Even though Perl has its own **sort** function, the external command is very good at dealing with large amounts of data in a memory-efficient manner.

```
use autodie;

# Open our external sort command.
open (my $sort_fh, "|sort");

# Our friends will be printed in sorted order.
foreach my $friend (qw/Jacinta Damian Kirrily Paul/) {
        print {$sort_fh} "$friend\n";
}
close $sort_fh;
```

If you’re interested in reading more about inter-process communication, including pipes, signals, sockets and the like, check out **perldoc perlipc**. In particular, bi-directional communication can be achieved with the `IPC::Open2` and `IPC::Open3` modules. The `Expect` module from the CPAN can be used to emulate a terminal if required.

Security issues

If you are intending to use information from the user when formulating your command you must take extra security precautions. When using two-argument open with pipes, Perl uses your system shell to run the command. If the user is allowed to specify any part of the command, they may be able to run code on your system that you did not expect. For example, consider the snippet from our earlier example:

```
use autodie;

# Process our command line arguments, and complain if we don’t
# have both a host and filename.  Note that we’re explicitly trusting our
# users to behave themselves with their inputs.

my ($host, $file) = @ARGV;

unless ($host and $file) {
        die "Usage: $0 host filename\n";
}

open (my $ssh_fh, "ssh $host cat $file |");
```

If our user passes in the following value for `$file`:

```
/etc/passwd; echo Hello World!
```

Then we will open a pipe to the result of `catting` `/etc/passwd` from the `$host` machine but we will also run `echo Hello World!` on the local machine.

If your program is running with elevated privileges, or is using input from a file generated from someone else, then you may have a command that’s much worse than `echo` being executed. Even without fears of malicious intent, we’d still like to avoid additional commands running by accident.

A more secure alternative is multi-argument `open`, available in versions of Perl from 5.8.0 onwards. This does not use the shell, but instead invokes the command directly and passes through the arguments individually. As ssh will allow us to pass in more than one command, separated by semi-colons as arguments, let’s consider the case where we are just `catting` a file on the local machine (although normally we’d just open it for reading):

```
use autodie;
my ($file) = @ARGV;
unless ($file) {
        die "Usage: $0 filename\n";
}
open (my $cat_fh, "-|", "cat", $file);
```

Now if we pass in `/etc/passwd; echo Hello World!` for our filename, `cat` will be given that string as the file to cat and will attempt to open the file by that name (literally); which probably doesn’t exist.

To open a command for writing (as we did in the `sort` example above) with multi-argument open, we’d use `|-` for our mode. The minus represents the position the command would sit relative to the pipe in two-argument open.

Multi-argument piped open does not work in any version of Perl on Microsoft Windows systems.

Prior to version 2.08, `autodie` did not support multi-argument piped `open`.

Exercises

1. Modify the sort example above (provided for you as `exercises/sort_starter.pl` in your exercises directory) to accept user input and print out the **sort**ed version.
2. Change your script to accept input from a file using `open()` (Answer: `exercises/answers/sort.pl`)
3. If you are using a Unix system: change your script to pipe its input through the **strings** command and then **sort**. Now if you specify a file that is not a text file, it will only sort and display printable strings. Try running this over `/usr/bin/perl`. (Answer: `exercises/answers/strings.pl`)

File locking

File locking can be achieved using the `flock()` function. This can be used to guard against race conditions or other problems which occur when two (or more) processes want to access the same file at the same time.

`flock()` is documented on page 714 (page 166, 2nd Ed) of the Camel book, or use **perldoc -f flock** to read the online documentation.

`flock` is Perl's portable file-locking mechanism, and works on most operating systems (and produces a fatal error on those which it does not). The locks set by `flock` are advisory only, which means that a process that chooses not to use `flock` can (and will) ignore any locks in place. `flock` can only lock entire files, not individual records. Depending upon your setup, `flock` may or may not work over NFS.

```
# using flock
use Fcntl qw(:flock);    # import LOCK_* constants

flock($fh, LOCK_EX);     # exclusive (write) access
flock($fh, LOCK_SH);     # shared (read-only) access
```

As `flock` only works on *filehandles*, instead of filenames, you have to open the file first *before* you try to lock it. It's important to make sure that you open the file for writing, if you intend to write to it, and that you don't clobber the contents of the file when doing so. This is a good use of `+<`. Closing a locked file releases any locks the process holds upon it. This is good because it means that if your process exits unexpectedly all locks it held are released and other processes may then go forward with their locks.

In the following example, we're locking a file before re-writing it. The exclusive lock stops any other process from holding a lock on the file while we perform our operations.

```
use Fcntl ’:flock’;             # import LOCK_* constants
use autodie;

# Open file for read and write
open my $file_fh, "+<", $file;

# Lock the file for writing (exclusive lock)
flock($file_fh, LOCK_EX);
```

```
# At this point we have exclusive access to the file.
# Wipe previous process’ details
truncate($file_fh, 0);

# Write to the file, or perform other operations as needed here...
print {$file_fh} $data;

close $file_fh;             # Closing the file releases the lock as well.
```

`flock` will wait indefinitely until the lock is granted, however it can return early if interrupted by a signal or other event. It’s important to ensure that flock returns *true* to be sure that you have the lock you requested. It is possible to make `flock` *non-blocking* as follows:

```
use Fcntl ’:flock’;             # import LOCK_* constants

flock($fh, LOCK_EX | LOCK_NB);     # non-blocking exclusive lock
flock($fh, LOCK_SH | LOCK_NB);     # non-blocking shared lock
```

All attempts to get a *non-blocking* lock return immediately with either *true* for success (the lock was obtained) or *false* for failure (the lock was not obtained).

For an excellent introduction on using `flock`, the slides from Mark Jason Dominus’ *File Locking Tricks and Traps* make excellent reading. They can be found at http://perl.plover.com/yak/flock/.

Handling binary data

If you are opening a file which contains binary data, you probably don’t want to read it in a line at a time using `while (<>) { }`, as there’s no guarantee that there will be any line breaks in the data, and we’ll probably end up with very uneven chunks.

Instead, we can use `read()` to read a certain number of bytes from a filehandle. However, before we do that, we should call the `binmode()` function on the filehandle, so that Perl knows that we’ll be dealing with a binary file. This means Perl won’t try to do any transformations of input based upon the operating system or locale where your program is running.

`binmode()` must be called on the filehandle before any corresponding file I/O. It’s best to call it immediately after you open the file.

You can learn more about `read()` by reading page 768 (page 202, 2nd Ed) of the Camel book or **perldoc -f read**.

You can learn more about `binmode()` by reading page 685 (page 147, 2nd Ed) of the Camel book, or **perldoc -f binmode**.

`read()` takes the following arguments:

- The filehandle to read from
- The scalar to put the binary data into
- The number of bytes to read

- The byte offset to start from (defaults to 0)

```
#!/usr/bin/perl
# Prints a random image
use strict;
use warnings;
use Local::File qw(random_image);
use autodie;
use CGI;

# Select a random image.
my $image = random_image();

# Open image file for reading
open(my $image_fh, "<", $image);

# Call binmode on our filehandles
binmode STDOUT;
binmode $image_fh;

# Print file headers
print CGI->header( -type => "image/jpg" );

# Print file contents to STDOUT
my $buffer;
while (read $image_fh, $buffer, 1024) {
        print $buffer;                     # Prints to STDOUT
}
close $image_fh;
```

Chapter summary

- Angle brackets <> can be used for simple line input. In scalar context, they return the next line; in list context, all remaining lines; the default filehandle is `STDIN` or any files mentioned in the command line (ie `@ARGV`).
- The `open()` and `close()` functions can be used to open and close files. Files can be opened for reading, writing, appending, read/write, or as pipes.
- File locking can be achieved using `flock()`.
- Binary data can be read using the `read()` function. The `binmode()` function should be used to ensure platform independence when reading binary data.

Chapter 21. System interaction

In this chapter...

In this chapter, we look at different ways to interact with the operating system. In particular, we examine the `system()` function, and the backtick command execution operator. We also look at security and platform-independence issues related to the use of these commands in Perl.

Platform dependency issues

Almost everything we’ve covered in our course so far works exactly the same on *nix, MacOS and Microsoft Windows systems. Once we start using system commands, we risk tying our code to a specific operating system and possibly even a particular set-up or machine.

Note that the examples given in this chapter will not work consistently on all operating systems. For example the use of `system()` calls or backticks with Unix-specific commands will not work under Windows NT, MacOS, etc. Slightly less obviously, the use of backticks on Windows NT can sometimes fail when the output of a command is sent explicitly to the screen rather than being returned by the backtick operation.

To understand more about how to make your Perl programs portable, read **perldoc perlport**.

system()

The `system()` function allows an external command to be executed. This command will inherit Perl’s standard filehandles (STDIN, STDOUT and STDERR) and have control of the console until it terminates. This means that `system()` can be used to launch interactive commands such as editors if desired. Perl will wait for the command specified to terminate before continuing:

```
# Open a file for the user to edit.
system(’vi somefile.txt’); # Or ’notepad somefile.txt’

# Read the contents of the file once the user has finished.
open(my $input_fh, ’<’, ’somefile.txt’) or die "$!";

# Cat all the .txt files to STDOUT
system(’cat *.txt’);
```

Error handling

Unlike all of Perl’s other built-in functions, `system` returns a true value upon failure, and a false value (zero) upon success. This is designed to mimic shell behaviour where zero means success and different failure conditions are signalled by varying exit values. For example **grep** returns 0 to indicate successful execution where lines were found that match the pattern; 1 to indicate successful execution but where lines were not found; and 2 to indicate that there was some sort of error. Thus to test `system` for failure we have to test for a true value, not a false value:

```
# Wrong!
system('/path/to/some/command') or die $!;

# Less wrong (but still wrong):
system('/path/to/some/command') and die $!;
```

If the command specified by `system()` could not be run, then `system` will return `-1` and the error message will be available via the special variable `$!`.

`$!` is not set if the command can be run but fails during runtime. The return status of the command can be found in the special variable `$?`, which is also the return value of `system()`. This value is a 16-bit status word which needs to be unpacked to be useful, as demonstrated in the example below.

```
# Recommended approach to testing success of system call
system("/path/to/some/command");
if ($?) {                     # A non-zero exit code means failure
                              # -1 means the program didn't even start!
        if($? == -1) {
                print "Failed to run program: $!\n";
        } else {
                print "The exit value was: " .  ($? >> 8) . "\n";
                print "The signal number that terminated the program was: "
                        . ($? & 127) . "\n";
                print "The program dumped core.\n" if $? & 128;
        }
}
```

While the above approach used to be the recommended way to process errors from `system`, modules now exist to do much of the hard work for you.

Security issues

Just like `open` the traditional form of calling `system` and `exec` have security issues due to shell expansion. For example consider the following code:

```
print "Please give me a file you want to see: ";

my $filename = <>;   # assume: $filename="fred; rm -rf /home/pjf;"

chomp($filename);
system("cat $filename");
```

In this case, due to shell expansion, the shell will receive the commands:

```
cat fred
rm -rf /home/pjf
```

and if our program had sufficient permissions to delete pjf's home directory, it would.

Multi-argument system

There is another, safer, form of `system` and `exec` that bypasses the shell. If you give `system` or `exec` a list it assumes that the first element is the command to execute, and every other element is an argument to that command. For example:

```
# Open a file for the user to edit.
system('vi', 'somefile.txt');
```

The arguments to the command are not passed to the shell, and so shell expansion will not occur. So:

```
system(’cat’, ’*.txt’);
```

will give the `"*.txt"` filename to `cat` rather than all files with a `.txt` extension. This is important in cases like the above where the command may be passed in from a user and contain shell characters.

In the above example, if the file "*.txt" does not exist then we’ll receive an non-zero return code and if we are doing our error handling correctly, we will report that properly.

It’s always possible to use Perl’s `glob` function to expand filenames for us, without shell intervention:

```
system( ’cat’, glob(’*.txt’) );
```

exec

`exec` works in a similar way to `system`, but on success your program is *replaced* with the literal specified. This means that `exec()` never returns on success. The `exec()` function is most useful when writing wrapper scripts that wish to establish a certain state before executing another program.

```
exec(’cat’, glob(’*.txt’));

print "This print statement should never be run!";
```

`exec` fails and returns false only if the command (in this case **cat**) does not exist. If the file given as the argument to **cat** does not exist, the user will receive **cat**’s error message and exit value.

IPC::System::Simple and autodie

You can read more about `IPC::System::Simple` at http://search.cpan.org/perldoc?IPC::System::Simple and `autodie` at http://search.cpan.org/perldoc?autodie.

Both the `IPC::System::Simple` module (available from the CPAN) and `autodie` can take the hard work out of checking the return value from system commands:

```
use IPC::System::Simple qw(system);

system("some_command");         # No need to check!
```

With `IPC::System::Simple` enabled, the `system` function will execute the command provided and check the result. If the command fails to start, dies from a signal, dumps core, or returns a non-zero exit status, then `IPC::System::Simple` will throw an exception with detailed diagnostics.

Capturing the exception

Unless you take steps to prevent it, a failure from this command will cause your program to die with an error. If you want to capture the error, you can do so:

```
# The 'eval' block allows us to capture errors, which are then placed
# in $@.  If any of the commands below fail, the 'eval' is exited
# immediately.  This means if we fail to backup the files, we won't
# delete them.

use IPC::System::Simple qw(system);

# Try to run these lines, catch any exceptions
eval {
        system('backup_files');
        system('delete_files');
};

# If we caught an exception, deal with it appropriately
if ($@) {
        warn "Error in running commands: $@\n";
}
```

The `Try::Tiny` module available at http://search.cpan.org/perldoc?Try::Tiny provides a cleaner syntax for exception handling, as well as protecting against numerous edge cases not discussed in this text.

With autodie

The `autodie` pragma uses `IPC::System::Simple` (only if installed) to provide the same changes to `system` but with lexical scope (until the end of the current block, file, or eval). The same code as in the previous example can also be written as:

```
# Try to run these lines, catch any exceptions
eval {
        use autodie qw(system);

        system('backup_files');
        system('delete_files');
};

# If we caught an exception, deal with it appropriately
if ($@) {
        warn "Error in running commands: $@\n";
}
```

autodie qw(:all)

When we write:

```
use autodie;
```

`autodie` makes most of Perl's built-ins succeed or die.

When we write:

```
use autodie qw(system);
```

`autodie` only turns on the succeed or die behaviour for `system` (and only if `IPC::System::Simple` is installed).

To make both autodie's usual list of Perl's built-ins and system succeed or die; we write:

```
use autodie qw(:all);
```

Of the IPC::System::Simple functions autodie only provides access to system function. If you need to access IPC::System::Simple's systemx, capture or capturex functions (discussed later) then you will need to use that module directly.

Allowable return values

As discussed earlier, some system commands may run successfully (that is, without error) but return a non-zero exit code to signal an unexpected situation. For example the system command grep returns 1 to indicate that it ran successfully, but that no lines matched the given pattern.

While being able to distinguish between success and matches and success and no matches is essential, by default this non-zero exit status would cause IPC::System::Simple (or autodie) to throw an exception and potentially kill your program.

Thus, we can tell IPC::System::Simple which return values are non-exceptional, as the first argument.

```
use IPC::System::Simple qw(system);

# Run a command, allowing any of 0, 1 or 2 as exit values
system( [0,1,2], "some_command");

# Run a command and capture its exit value:
my $exit_value = system( [0,1,2], 'some_command');
```

IPC::System::Simple and the shell

Just like the built-in system, IPC::System::Simple's system command uses the standard shell when running a single command, or invokes the command directly when called in a multiple argument fashion:

```
use IPC::System::Simple qw(system);

# Run 'cat *.txt' via the shell.
system('cat *.txt');

# Run 'cat' on the file called '*.txt', bypassing the shell.
system('cat', '*.txt');

# Run 'cat' on all files matching '*.txt', bypassing the shell.
system('cat', glob('*.txt'));
```

systemx

The IPC::System::Simple module also provides a systemx() command for running commands, but which *never* invokes the shell, even when called with a single argument.

```
use IPC::System::Simple qw(systemx);

# Run $program, always bypassing the shell, even if @options is empty.
systemx($program, @options);
```

Exercises

*nix exercises

1. Write a script to ask the user for a username on the system, then perform the **finger** command to see information about that user. (Answer: `exercises/answers/finger.pl`)

MS Windows exercise

1. Write a script to ask the user for a filename on the system. Open the nominated file in Notepad using **system**. (Answer: `exercises/answers/notepad.pl`)

Using backticks

Single quotes can be used to specify a literal string which can be printed, assigned to a variable, et cetera. Double quotes perform interpolation of variables and certain escape sequences such as `\n` to create a string which can also be printed, assigned, etc.

A new set of quotes, called *backticks*, can be used to interpolate variables then run the resultant string as a shell command. The output of that command can then be printed, assigned, and so forth.

Backticks are the backwards-apostrophe character (‘) which appears below the tilde (~), next to the number `1` on most keyboards.

Just as the `q()` and `qq()` functions can be used to create a single or double quoted string to save you from having to escape quotes that appear within a string, the equivalent function `qx()` can be used for backticks.

In this course we tend to use `qx()` because it’s much harder to confuse `qx()` with plain old single quotes. Using `qx()` also avoids the problem that in some font sets both single quotes and backticks look exactly the same.

Backticks and the `qx()` function are discussed in the Camel book on page 80 (pages 41 and 52, 2nd Ed) or in **perldoc perlop**.

Backticks vs system()

Backticks are different to the `system()` command, in that they capture the output of the command they execute, as opposed to passing it through to the user.

When called in a scalar context backticks return the output of the command they execute as a string with possibly embedded newlines. When called in a list context, the output is returned as a list with each separate line of output being a new list element.

```
#!/usr/bin/perl
use strict;
use warnings;

# Backticks capture the output of the process they run.  Here,
# we capture the output of the echo command.

my $greeting = qx(echo Hello World);

# $greeting now contains the string "Hello World\n"

# System runs a command without capturing the output, instead it’s
# passed straight through.  The following line uses the echo command
# to print a greeting.

system("echo Hello World");
```

The return status of commands called by using backticks can be determined by examining `$?` in the same way as the `system()` example above.

Security issues

Backticks *always* use the shell. It’s very important to make sure your data is always properly sanitised before using the backticks operator. Perl’s taint mode (see later this chapter) can assist with this. `IPC::System::Simple` also provides an alternative (see below).

Capturing with IPC::System::Simple

`IPC::System::Simple` also provides the `capture()` command as an alternatives to backticks. It works in an identical fashion to backticks, but throw exceptions if the command fails, and can be passed multiple arguments to avoid the shell.

```
use IPC::System::Simple qw(capture);

# Runs the ’hostname’ command and captures the output.  If the command
# fails, or cannot be found, dies with an error.

my $cmd_output = capture(’hostname’);

# Run the ’cat’ command on all files matching ’*.txt’, bypassing the
# shell.  The results are placed into @lines.

my @lines = capture(’cat’, glob(’*.txt’));

# Run a command, capture its output, and insist it returns an exit
# status of only 0, 1, or 2:

my $cmd_output2 = capture( [0,1,2], ’some_command’);
```

The module also provides a `capturex()` call which works just like backticks, but which never invokes the shell, even when called with a single argument.

IPC::System::Simple's capture() will *never* use the shell under Microsoft Windows - even if passed a single argument. Thus things which are shell built-ins, such as dir, will not work within a call to capture or capturex.

Exercises

*nix exercises

1. Modify your earlier finger program to use backticks or qx() instead of system() (Answer: exercises/answers/backtickfinger.pl)
2. Change it to use capture with a single argument instead. What do you have to do to make it work with multiple arguments (ie without going via the shell)? (Answer: exercises/answers/capturefinger.pl)
3. The Unix command **whoami** gives your username. Since most shells support backticks, you can type **finger 'whoami'** to finger yourself. Use shell backticks inside your qx() statement to do this from within your Perl program. (Answer: exercises/answers/qxfinger2.pl)
4. The **/sbin/ifconfig** command tells you information about your system's network interfaces. Find your machine's IP address in this output (ask your trainer if you need a hand). Using backticks, qx() or capture and a regular expression, print out the IP address from **/sbin/ifconfig**. (Answer: exercises/answers/ifconfig.pl

MS Windows exercises

1. Modify your earlier program to take a directory path from the user. Use backticks or qx() to execute the **DIR** command on that path and list out the files in that directory. (Answer: exercises/answers/backtickdir.pl) Note: IPC::System::Simple::capture will not work for this exercise.
2. Run **ipconfig** on your DOS command line. Identify your machine's IP address. Using backticks, qx() or capture and a regular expression, print out the IP address from **ipconfig**. (Answer: exercises/answers/ipconfig.pl)
3. Time permitting: reverse sort the directory listing contents.

Using tainted data (Perl's security model)

This section is not intended as a comprehensive guide to Perl security, rather it is here to show some of the in-built security features that Perl has available. Even perldoc perlsec does not give you the whole picture, it just gives you some hints.

> The ability to write secure programs is one that is learnt over many years of experience. It’s always a good idea to have someone well rehearsed in security and your programming environment to audit your code in case you have missed anything. As well as training, Perl Training Australia also offers security and privacy auditing services.
>
> Perl Training Australia offers a course in Perl Security that covers many common attacks and mistakes, and how they can be prevented in Perl.

Insecure code

Many of the examples given above can result in major security risks if the commands executed are based on user input. Consider the example of a simple finger program which asked the user who they wanted to finger:

```
#!/usr/bin/perl
use strict;
use warnings;

print "Who do you want to finger? ";
my $username = <STDIN>;
print qx(finger $username);
```

Imagine if the user’s input had been `pjf; cat /etc/passwd`, or worse yet, `pjf; rm -rf /`. The system would perform both commands as though they had been entered into the shell one after the other.

Which program?

A further, not so obvious problem, can be seen when we ask "Which `finger` program are we calling?". If our program caller has changed our `$ENV{PATH}` then it is very possible that it’s not the usual system `finger` found in `/usr/bin/`. It could instead be a malicious `finger` program designed to exploit our program’s privileges.

Explicitly setting $ENV{PATH}

To ensure we call the correct program, code in taint mode must explicitly set the `PATH` variable (found in `$ENV{PATH}`) to something safe (like `/usr/bin`). This variable affects where the shell looks for other executable programs.

We have to set a safe value for `$ENV{PATH}` because this value can be changed by the user in their environment before running the Perl script. If the user sets their `PATH` to `/home/pjf/bin` then we’d run the `/home/pjf/bin/finger` command rather than the `/usr/bin/finger` command.

For safety’s sake, taint checking in Perl always assumes that the `PATH` environment variable has been tampered with by the user.

The correct values to add to your `PATH` depend on your operating system, individual machine set-up and where the executables you intend to call are located. Some common values are shown below:

```
# Some *nix systems (paths are separated by :s)
$ENV{PATH} = "/bin:/usr/bin:/usr/local/bin";

# Microsoft Windows (paths are separated by ;s)
$ENV{PATH} = ’C:\Windows;C:\Windows\System32’;
```

Turning on taint

Perl's `-T` flag can be used to check for unsafe user inputs.

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

`-T` stands for "taint checking". Data input by the user is considered "tainted" and until it has been modified by the script, may not be used to perform shell commands or system interactions of any kind. This includes system interactions such as `open()`, `chmod()`, and any other built-in Perl function which interacts with the operating system.

If you've been calling your Perl program from the command line with **perl *`program.pl`*** you'll be told that you're turning taint checking on too late, even if you've put it in your shebang line.

This is because Perl wants to know that you want to use taint checking as soon as possible. The way to fix this is to include the `-T` option in your call, so: **perl -T *`program.pl`***.

Documentation for taint checking can be found by reading the **perldoc perlsec**, or on pages 557-568 (page 356, 2nd Ed) of the Camel book.

In versions of Perl prior to 5.8.0 files opened for both reading and writing using "`+<`" were not checked for tainted filenames.

Taint checking will not occur on filenames where the file is only being opened for reading. This is due to historical reasons. Good programming practice would have you untaint these filenames anyway.

SETUID scripts automatically run with taint checking turned on for your own protection.

Under Perl 5.8.0 and above, there is also the `-t` switch, which causes tainted operations to generate warnings instead of errors. This is no substitute for real taint checking, but can be useful if you're trying to lock down legacy code and see which areas require attention.

Cleaning tainted data

The only thing that will clear tainting is referencing substrings from a regexp match. Here's an example.

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

$ENV{PATH} = "/bin:/usr/bin"; # Taint requires we set our path.

print "Who do you want to finger?\n";
my $username = <STDIN>;
chomp($username);

# Check $username looks right.  If so, use the *captured* value with
# finger.
if ( $username =~ /^(\w{1,20})$/ ) {

       # $1 is the contents of the first set of
       # parentheses in the regexp.

       print qx(finger $1);
}
else {
       print "That was not a valid username!\n";
}
```

Make sure you remember to check that the regular expression to untaint your variable succeeded. In the case above we only have one regular expression, so `$1` will either be set by the match or will be undefined. Nevertheless we still explicitly tested the match for success. This means that our code won't break if we add any regular expressions before the code used above.

You can also untaint data by capturing the match in a list context:

```
# Check $username to make sure it matches the pattern
# Condition will fail if no match, or set $safeuser to safe value ($1) on
# success
if ( my ($safeuser) = ($username =~ /^(\w{1,20})$/) )
       print qx(finger $safeuser);
}
else {
       print "That was not a valid username!\n";
}
```

Exercise

1. Ask the user for a filename, open the file and write a short message to it. Turn on taint checking and try running your script. Turn on taint, untaint the filename and ensure your code works. (Answer: `exercises/answers/taintfile.pl`)

   Hint: Filenames consist of 1 or more alphanumeric characters, hyphens, underscores and dots. Try these as your first pass. Later, depending on how advanced you want your answer to be, filenames may also consist of directory separators, spaces and possibly some other characters.

2. Turn taint mode on for your `sort_starter.pl` exercise from the *File I/O* chapter. What do you need to add to make this program work now?

Chapter summary

- The `system()` function can be used to perform system commands. `$!` is set if any error occurs.
- The backtick operator can be used to perform a system command and return the output. The `qx()` quoting function/operator works similarly to backticks.
- The above methods may not result in platform independent code.
- Data input by users or from elsewhere on the system can cause security problems. Perl's `-T` flag can be used to check for such "tainted" data
- Tainted data can only be untainted by referencing a substring from a pattern match.

Chapter 22. Practical exercises

About these exercises

These exercises are designed to complement the existing course exercises and provide a broader coverage of Perl. They are designed to range in level of difficulty and may require skills we haven't yet covered in the course. When you find that you don't have the knowledge to solve a problem, feel free to move onto another puzzle instead.

Although these exercises should be fun to work on, please work first on the course exercises you've been assigned. The course exercises are designed to enhance your understanding of the material just covered, and are essential in consolidating your understanding of Perl.

Palindromes

A palindrome is an integer or string which reads the same both forwards and backwards. For example 1441, and "Hannah". If we allow multi-word palindromes we can also have sentences such as "Able was I ere I saw Elba". Each of these are "true" palindromes as (ignoring case) each string reads exactly the same forwards as it does backwards.

If we extend the definition of a palindrome such that any sequence of word characters is considered, regardless of spacing and punctuation we can get a much wider range. For example each of the following are palindromes: "race car", "Madam, in Eden I'm Adam", "Was it a cat I saw?", "Did I do, O God, did I as I said I'd do? Good, I did."

1. Write a program which detects whether a string is a true palindrome irrespective of case.
2. Extend your program to detect whether a string is a palindrome by ignoring capitalization and spacing. If we haven't covered regular expressions yet, you may find the `split` and `join` functions handy.
3. Now allow for punctuation. You'll probably want to use a regular expression for this task.
4. In assembler the solution to this problem would be to walk two pointers along the string starting at opposite ends and comparing character by character. Punctuation would be handled by incrementing the pointers at each point until they reached the next word character. Comparison would stop with failure if two characters were unequal, and with success if the pointers reached the same location or passed each other. This is called an "in place comparison".

   Using either `substr` or using split then walking over an array; write a program which determines if a string is a palindrome using in-place comparison only.

Hangman

The game of hangman is a common pastime for young children. The game master picks a word and the player has to guess the word by choosing letters that may be in that word. If the letter is correct the game master writes the letter into all the correct positions of the word. If the guess is wrong, more of the hanged man is drawn. In this exercise we won't draw the hangman, but we'll keep track of the guesses remaining and the letters guessed.

1. Write a program which reads in the contents of a text file and randomly picks a word. You can read a file passed in on the command line with `while(<>)`.

   Make sure that words of less than 4 letters and those containing punctuation are not chosen. If you're working on a Unix-like machine you may want to pick a word from the `/usr/share/dict/words` file.

2. Extend your program to allow the user to play hangman. At the start of each turn, report to the user the number of letters the selected word has, which letters they have already guessed (and their locations if successful) and the number of guesses remaining. For example your output may look like:

   ```
   e _ e _ _ _ _ t  (8 letters).  Guesses (e, t, s).
                     7 guesses remaining.
   ```

3. Accept options on the command line for the maximum number of guesses, and the minimum and maximum word length. You may find `Getopt::Std` useful for this.

4. If the player wins, ask them for their name and add their name to a high-score table. This table should list the players name, the length of the word and the number of wrong guesses they made. Write the information out to a file so that you can display all the high scores (sorted by word length and wrong guesses) at the successful completion of each game. You may find `Storable` to be helpful.

   An example high score table might look like:

   ```
   Name                    Word length     Mistakes
   ------------------------------------------------
   Paul Fenwick                10            3
   Jacinta Richardson          10            5
   Jacinta Richardson           9            2
   Paul Fenwick                 8            4
   ```

Chapter 23. Conclusion

Where to now?

To further extend your knowledge of Perl, you may like to:

- Work through the material included in the appendices of this book.
- Visit the websites in our "Further Reading" section (below).
- Follow some of the URLs given throughout these course notes, especially the ones marked "Readme".
- Install Perl on your home or work computer.
- Practice using Perl from day to day.
- Join a Perl user group such as Perl Mongers (http://www.pm.org/).
- Join an on-line Perl community such as PerlMonks (http://www.perlmonks.org/).
- Extend your knowledge with further Perl Training Australia courses such as:
  - Web Development with Perl
  - Database Programming with Perl
  - Perl Security
  - Object Oriented Perl

  Information about these courses can be found on Perl Training Australia’s website (http://www.perltraining.com.au/).

Further reading

Books

- Larry Wall, Tom Christiansen and Jon Orwant, *Programming Perl* (3rd Ed), O’Reilly and Associates, 2000. ISBN 0-596-00027-8
- Tom Christiansen and Nathan Torkington, *The Perl Cookbook*, O’Reilly and Associates, 1998. ISBN 1-56592-243-3.
- Jeffrey Friedl, *Mastering Regular Expressions*, O’Reilly and Associates, 1997. ISBN 1-56592-257-3.
- Joseph N. Hall and Randal L. Schwartz *Effective Perl Programming*, Addison-Wesley, 1997. ISBN 0-20141-975-0.
- Damian Conway, *Perl Best Practices*, O’Reilly and Associates, 2005. ISBN 0-59600-173-8.

Online

- The Australian Perl Portal (http://www.perl.net.au/)
- Perl 5 Wiki (http://perlfoundation.org/perl5)
- Perl Mongers Perl user groups (http://www.pm.org/)
- PerlMonks online community (http://www.perlmonks.org/), in particular check out the tutorials (http://perlmonks.org/?node=Tutorials)
- Comprehensive Perl Archive Network (http://search.cpan.org)
- The Perl homepage (http://www.perl.com/)
- The Perl Directory (http://www.perl.org/)
- Perl Quality Assurance Projects (http://qa.perl.org/)
- Perl Critic homepage (http://perlcritic.com/)

Appendix A. Advanced Perl variables

In this chapter...

In this chapter we will explore Perl’s variable types a little further. We’ll look at hash slices and cool ways to assign values into and from arrays and hashes. But first we’ll look at how we can make quoting a little nicer.

Quoting with `qq()` and `q()`

Using double quotes or single quotes when quoting some strings can result in lots of character escaping. Which quotes are best for quoting the following paragraph?

```
Jamie and Peter’s mother couldn’t drive them to the show.
"How are we going to get there?" Jamie asked.
"We could ride our bikes", Peter suggested.
But Peter’s bike had a flat tyre.
```

If we use double quotes it comes out looking like this:

```
print "Jamie and Peter’s mother couldn’t drive them to the show.
\"How are we going to get there?\" Jamie asked.
\"We could ride our bikes\", Peter suggested.
But Peter’s bike had a flat tyre.";
```

but that’s just ugly. Single quotes aren’t much better:

```
print ’Jamie and Peter\’s mother couldn\’t drive them to the show.
"How are we going to get there?" Jamie asked.
"We could ride our bikes", Peter suggested.
But Peter\’s bike had a flat tyre.’;
```

In order to encourage beautiful code that you can be proud of, Perl allows you to pick your own quote operators, when you need to, by providing you with `q()` and `qq()`. `q()` represents single quotes and `qq()` represents double quotes. Note that the same rules apply for each of these quoting styles as for their more common equivalents: `qq()` allows variable interpolation and control character expansion (such as the newline character) whereas `q()` does not. These are often called "pick your own quotes" or "roll your own quotes".

Using pick your your own quotes, quoting the above paragraph becomes easy:

```
qq(Jamie and Peter’s mother couldn’t drive them to the show.
"How are we going to get there?" Jamie asked.
"We could ride our bikes", Peter suggested.
But Peter’s bike had a flat tyre.);
```

You may use any non-whitespace, non-alphanumeric character as your delimiters. Pick one not likely to appear in your string. Note that things that look like they should match up do. So `(` matches `)`, `{` matches `}` and finally < matches >. There are some illustrated below.

```
print q/Jamie said "Using slashes as quoting delimiters is very common."/;
print q(Jamie said "You should always watch your quotes!");
print qq!Jamie said "$these are Paul’s favourite quotes". (He was wrong).\n!;
print qq[Jamie said "Perl programs ought to start with #!"\n];
print qq#Jamie said "My favourite regexp is ’/[jamie]*/i;’"\n#;
```

> ☞ If you use matching delimiters around your quoted text Perl will allow you to include those delimiters in your quoted text if they are also paired.
>
> ```
> print qq(There was a (large) dog in the yard\n);      # This will work
> ```
>
> If the delimiters within your quoted text are not paired, this will result in errors.
>
> ```
> print qq< 1 + 4 < 10 >;                               # This will not work
> ```
>
> The problem with the last example is that Perl assumes that the closing > is paired with the second < and waits to see a later > to close the string.

A different way of quoting strings are `HERE` documents. These can sometimes be confusing for the reader, and usually pick your own quotes will be clearer. We cover `HERE` documents here for the sake of completeness, and because they are still very common in older code. If you've done a lot of shell programming you may recognise this construct. `HERE` documents allow you to quote text up to a certain marking string. For example:

```
print << "END";
I can print any text I want to put here without
fear of "weird" things happening to it.    All
punctuation is fine, unlike roll-your-own quotes,
where you have to pick some kind of punctuation to
delimit it.    Here, we just have to make sure that
the word, up there (next to print) does not appear
in this text, on a line by itself and unquoted.
Otherwise we terminate our text.
END
```

The quoting style used in HERE documents is whatever you quote the terminating word with next to the print statement (in this case double quotes). Using double quotes results in variable interpolation, whereas using single quotes results in no variable interpolation.

Exercises

1. Experiment with using `q()` and `qq()` to print the following string:

   `\/<+c&b^$!a@_#‘*’"~{ [()] }~"’*‘#_@a!$^b&c+>\/`
   you'll find this string in the file: `exercises/quoteme.pl`

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

Scalars in assignment

You may find yourself wishing to declare and initialise a number of variables at once:

```
my $start = 0;
my $end = 100;
my $mid = 50;
```

but you don't want to take up three lines to do it in. Perl lets you do the following:

```
my ($start, $end, $mid) = (0, 100, 50);
```

which says create the variables `$start`, `$end` and `$mid` and assign them values from the list on the right dependent on their list position. You'll see this kind of thing all the time. If the list on the right is longer than the list on the left, the extra values are ignored. If the list on the left is longer than the list on the right, the extra variables get no value.

```
my ($a, $b, $c) = (1, 2, 3, 4, 5);   # $a = 1, $b = 2, $c = 3.
                                     # values 4 and 5 are ignored.
my ($d, $e, $f, $g) = (1, 3, 5);     # $d = 1, $e = 3, $f = 5.
                                     # $g gets no value.
```

If the variables are already declared with `my` elsewhere, you can still use the above method to assign to them.

```
($a, $b, $c) = (1, 4, $d);           # $a = 1, $b = 4, $c = $d.
```

In fact, this gives us a very simple way to swap the values of two variables without needing a temporary variable:

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

! You'll notice above that in all the examples we've grouped our lists within parentheses. These parentheses are required.

Arrays in assignment

Just as we could assign a list of values to a list of scalars, we can assign elements from arrays to a list of scalars as well. Once again if we provide more values on the right than we provide variables on the left, the extra ones are ignored. If we provide more variables on the left than values on the right, the extra variables are given no value.

```
my ($fruit1, $fruit2, $fruit3) = @fruits;         # assign from array
my ($number1, $number2) = @magic_numbers[-2, -1]; # assign from array slice
my @short = (1,2);
my ($a, $b, $c) = @short;                         # $c gets no value
($a, $b) = @random_scalars;                       # changes $a and $b.
```

Sometimes we would like to make sure that we get enough values in our list to initialise all of our variables. We can do this by supplementing our list with reasonable defaults:

```
my @short = (1, 2);
my ($a, $b, $c, $d, $e, $f) = (@short, 0, 0, 0, 0, 0, 0);
```

this way, even if `@short` is completely empty we know that our variables will all be initialised.

So what happens if you put an array on the left hand side? Well, you end up with an array copy.

```
my @other_fruits = @fruits;        # copies @fruits into @other_fruits
my @small_fruits = @fruits[0..2];  # copies apples, oranges and guavas into
                                   # @small_fruits.
```

What happens if you put two arrays on the left and two on the right? Do you end up with two array copies? Can you use this to swap the contents of two arrays? Unfortunately no.

```
(@a, @b) = (@c, @d);                # Does @a = @c, @b = @d ?   No.
                                    # Instead:
                                    # @a = @c and @d joined together
                                    # @b is made empty

(@a, @b) = (@b, @a);                # Are array contents swapped? No.
                                    # Instead:
                                    # @a becomes @b and @a joined together
                                    # @b is made empty.
```

When two arrays are put together into a list, they are "flattened" and joined together. This is great if you wish to join two arrays together:

```
my @bigger = (@small1, @small2, @small3);         # join 3 arrays together
```

but a bit awkward if you were hoping to swap their contents. To get two array copies or to swap the contents of two arrays, you're going to have to do it the long way.

Hash slices

Hash slices are used less frequently than array slices and are usually considered more confusing. To take a hash slice we do the following:

```
# Our hash
my %people = (
        James => 30,
        Ralph => 5,
        John => 23,
        Jane => 34,
        Maria => 26,
        Bettie => 29
);

# An array (some of the people in %people)
my @friends = qw(Bettie John Ralph);

# Taking a hash slice on the %people hash using the array @friends to
# give us the keys.
my @ages = @people{@friends};               # @ages contains: 29, 23, 5

my @ages_b = @people{qw(Bettie John Ralph)}; # essentially the same as above
```

You'll notice that when we did the hash slice we used an `@` symbol out the front rather than a `%` sign. This isn't a typographical error. The reason we use an `@` sign is because we're expect a list (of values) out. Perl knows that we're looking at the hash called `%people` rather than any array called `@people` because we've used curly braces rather than square brackets.

Searching for items in a large list

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 black brown purple blue
                 white grey fawn lemon apricot cream indigo);

# A list of colours that we’re interested in which may or not be in
# @colours
my @find = qw(red blue green peach);

# We’ll use this hash to remember all the values in @colours for fast
# look-up
my %check;
@check{@colours} = ();  # set all values in %check from the keys in
                        # @colours to have the undefined value (but
                        # exist in the hash).

# We now look for @find in %check rather than @colours.  This very fast.
foreach my $wanted (@find) {
        if( exists( $check{$wanted} ) ) {
                print "$wanted is in \@colours array\n";
        }
        else {
                print "$wanted is not in \@colours\n";
        }
}
```

Unique values

We can use the fact that hash keys are unique to get the unique elements from a hash.

```
my @duplicates = (1, 1, 2, 3, 4, 5, 5, 6, 7, 7, 8, 9, 2, 3, 3,);

my %unique;
@unique{@duplicates} = ();

print "Unique elements are ", join(" ", sort keys %unique), "\n";
```

Keep in mind that due to the properties of hashes, the order of the original duplicates array will not be preserved by this technique.

Exercise

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 and a foreach loop to print out the unique values of the above list in first-seen order (ie: one two three four five).

Answers for the above questions can be found in `exercises/answers/duplicates.pl`.

Hashes in assignment

Assignment from hashes is a little different to assignment from arrays. If you try the following:

```
my ($month1, $month2) = %monthdays;
```

you won't get the names of two months. When a hash is treated as a list it flattens down into a list of key-value pairs. This means that `$month1` will certainly be the name of a month, but `$month2` will be the number of days in `$month1`.

To get all they keys of a hash we use the `keys` function. If we wanted two of these we can do the following:

```
my ($month1, $month2) = keys %monthdays;
```

To get two values from this hash (which would match the keys we've pulled out above) we use the `values` function.

```
my ($days1, $days2) = values %monthdays;
```

As the `values` function only returns the values inside the hash and we cannot easily determine from a value which key it had, using the `values` function loses information. Usually the values in a hash are accessed through their keys:

```
my $days1 = $monthdays{January};
my $days2 = $monthdays{February};

my ($days1, $days2) = @monthdays{qw/January February/};  # a shorter way
                                                          # if we want a few
```

We can use the fact that hashes flatten into lists when used in list context to join hashes together.

```
my %bigger = (%smaller, %smallest);
```

Note, however, that because each hash key must be unique that this may result in some data loss. When two hash keys clash the earlier one is over written with the later one. In the case above, any keys in `%smaller` that also appear in `%smallest` will get the values in `%smallest`. This is great news if you have a hash of defaults you want to use if any values are missing.

```
my %defaults = (
        name => "John Doe",
        address => "No fixed abode",
        age => "young",
);

my %input = (
        name => "Paul Fenwick",
        address => "c/o Perl Training Australia",
);

%input = (%defaults, %input);       # join two hashes, replacing defaults
                                    # with provided values
                                    # age was missing; gets set to "young"
```

To copy a hash you can just assign its value to the copy hash. However, attempts to perform a double copy in one step or to swap the values of two hashes without a temporary hash result in the same issues as with arrays due to list flattening.

Chapter summary

- Using `q()` and `qq()` allows the programmer to chose quoting characters other than `"` and `'`.
- Perl allows paired delimiters to also appear in the quoted text when using `q()` and `qq()` so long as those characters are also paired.
- Perl allows programmers to initialise scalar variables from lists and to provide less or more values than required if desired.
- You can swap the value of two scalar variables by assigning their values to each other in a list assignment.
- Arrays can be copied by assigning one array to another.
- Arrays flatten to one big list when combined in list context.
- Hash slices allow us to access several values from a hash in one step.
- Hashes can be copied by assigning one hash to another.

Appendix B. Complex data structures

References are most often used to create complex data structures. Since references are scalars, they can be used as values in both hashes and arrays. This makes it possible to create both deep and complex multi-dimensional data structures. We'll cover some of these in further detail in this chapter.

Complex data structures are covered in detail in chapter 9 (chapter 4, 2nd Ed) of the Camel book.

Arrays of arrays

The simplest kind of nested data structure is the two-dimensional array or matrix. It's easy to understand, use and expand.

Creating and accessing a two-dimensional array

To create a two dimensional array, use anonymous array references:

```
my @AoA = (
        [qw(apple orange pear banana)],
        [qw(mouse rat hamster gerbil rabbit)],
        [qw(camel llama panther sheep)],
);

print $AoA[1]->[3];       # prints "gerbil"
```

The arrow is optional between brackets or braces so the above access could equally well have been written:

```
print $AoA[1][3];
```

Adding to your two-dimensional array

There are several ways you can add things to your two-dimensional array. These also apply to three and four and five and n-dimensional arrays. You can push an anonymous array into your array:

```
push @AoA, [qw/lions tigers bears/];
```

or assign it manually:

```
$AoA[5] = [qw/fish chips vinegar salt pepper-pop/];
```

You can also add items into your arrays manually:

```
$AoA[0][5] = "mango";
```

or by pushing:

```
push @{$AoA[0]}, "grapefruit";
```

You’re probably wondering about why we needed the curly braces in our last example. This is because we want to tell Perl that we’re looking at the element `$AoA[0]` and asking it to deference that into an array. When we write `@$AoA[0]` Perl interprets that as `@{$AoA}[0]` which assumes that `$AoA` is a reference to an array we’re trying to take an array slice on it. It’s usually a good idea to use curly braces around the element you’re dereferencing to save everyone from this confusion.

Printing out your two-dimensional array

Printing out a single element from your two-dimensional array is easy:

```
print $AoA[1][2];            # prints "hamster"
```

however, if you wish to print our your data structure, you can’t just do this:

```
print @AoA;
```

as what you’ll get is something like this:

```
ARRAY(0x80f606c)ARRAY(0x810019c)ARRAY(0x81001f0)
```

which are stringified references. Instead you’ll have to create a loop to print out your array:

```
foreach my $list (@AoA) {
        print "@$list";
}
```

Hashes of arrays

Arrays of arrays have their uses, but require you to remember the row number for each separate list. Hashes of arrays allow you to associate information with each list so that you can look up each array from a key.

Creating and accessing a hash of arrays

To create a hash of arrays create a hash whose keys are anonymous arrays:

```
my %HoA = (
        fruits =>  [qw(apple orange pear banana)],
        rodents => [qw(mouse rat hamster gerbil rabbit)],
        books =>   [qw(camel llama panther sheep)],
);

print $HoA{rodents}[3];         # prints "gerbil"
```

Adding to your hash of arrays

Adding things to your hash of arrays is easy. To add a new row, just assign an anonymous array to your hash:

```
$HoA{oh_my} =  [qw/lions tigers bears/];
```

To add a single element to an array, either add it in place or push it on the end:

```
$HoA{fruits}[4] = "grapefruit";
push @{$HoA{fruits}}, "mango";
```

Once again you’ll notice that we needed an extra set of curly braces to make it clear to Perl that we wanted `$HoA{fruits}` dereferenced to an array.

Printing out your hash of arrays

Printing out a single element from your hash of arrays is easy:

```
print $HoA{fruits}[2];          # prints "pear"
```

Printing out all the element once again requires a loop:

```
foreach my $key (keys %HoA) {
        print "$key => @{$HoA{$key}}\n";
}
```

Arrays of hashes

Arrays of hashes are particularly common when you have number of ordered records that you wish to process sequentially, and each record consists of key-value pairs.

Creating and accessing an array of hashes

To create an array of hashes create an array whose values are anonymous hashes:

```
my @AoH = (
        {
                name => "John",
                age => 31,
        },
        {
                name => "Mary",
                age => 23,
        },
        {
                name => "Paul",
                age => 27,
        },
);

print $AoH[2]{name};            # prints "Paul"
```

Adding to your array of hashes

To add a new hash to your array, add it manually or push it on the end. To add an element to every hash use a loop:

```
$AoH[3] = {                             # adding a new hash manually
                name => "Jacinta",
                age => 26,
          };

push @AoH, {                            # pushing a new hash on to the end
                name => "Judy",
                age => 47
           };

$AoH[0]{favourite_colour} = "blue";     # adding an element to one hash

foreach my $hashref (@AoH) {            # adding an element to every hash
        $hashref->{language} = "Perl";
}
```

Printing out your array of hashes

To print a array of hashes we need two loops. One to loop over every element of the array and a second to loop over the keys in the hash:

```
foreach my $hashref (@AoH) {
        foreach $key (keys %$hashref) {
                print "$key => $hashref->{$key}\n";
        }
}
```

Hashes of hashes

Hashes of hashes are an extremely common sight in Perl programs. Hashes of hashes allow you to have a number of records indexed by name, and for each record to contain sub-records. As hash lookups are very fast, accessing data from the structures is also very fast.

Creating and accessing a hash of hashes

To create a hash of hashes, assign anonymous hashes as your hash values:

```
my %HoH = (
        Jacinta => {
                        age => 26,
                        favourite_colour => "blue",
                        sport => "swimming",
                        language => "Perl",
                   },
        Paul => {
                        age => 27,
                        favourite_colour => "green",
                        sport => "cycling",
                        language => "Perl",
                },
        Ralph => {
                        age => 7,
                        favourite_colour=> "yellow",
                        sport => "little athletics",
                        language => "English"
                 },
);

print $HoH{Ralph}{sport};       # prints "little athletics"
```

Adding to your hash of hashes

```
$HoH{Ralph}{favourite_food} = "Tomato sauce";   # adding to Ralph’s hash

$HoH{Tina} = {                                  # adding a new person hash
                age => 19,
                favourite_colour => "black",
                sport => "tai chi",
};
```

Printing out your hash of hashes

Once again, to print out a hash of hashes we’ll need two loops, one for each key of the primary hash and the second for each key of the inner hash.

```
foreach my $person ( keys %HoH ) {
        print "We know this about $person:\n";
        foreach $key ( keys %{ $HoH{$person} } ) {
                print "${person}’s $key is $HoH{$person}{$key}\n";
        }
        print "\n";
}
```

More complex structures

Armed with an understanding of the nested data structures we’ve just covered you should be able to create the best data structure for what you need. Perhaps you need a hash of hashes but where some of your values are arrays. This should pose no problems. Perhaps you want an array of hashes of arrays? This too should be easy.

Appendix C. Directory interaction

In this chapter...

In this chapter, we learn how to work with directories in various ways.

Reading a direction with glob()

Like all functions, you can read more about `glob` using `perldoc -f glob`.

The easiest way to interact with a directory is by using Perl's `glob` function. This accepts a glob pattern (such as `*.txt`) and returns all the files which match that pattern. In a list context all the files are returned and in a scalar context the next file is returned.

Perl's globs operate the same way as they do in the UNIX C-shell. Don't worry if you don't know C-shell, the basic pattern matching operators (such as `*` and `?`) have the same behaviour as just about any other shell that you may have used.

```
# Get all the files matching *.txt and process them one by one.
use autodie;

while (my $filename = glob("*.txt")) {
        open (my $in_fh, "<", $filename);

        # Read from the file

        close $in_fh;
}

# Get all the files matching *.pl and put them in our array
my @files = glob("*.pl");
```

When using `glob`, you can combine more than one pattern in order to get a wider selection of files. For example, to get all the `*.txt` and `*.pl` files you can write:

```
glob("*.txt *.pl");
```

Exercises

1. Use the file globbing function to find all Perl scripts in your current directory and print out their names (assuming they are named in the form `*.pl`) (Answer: `exercises/answers/findscripts.pl`)
2. Use the above example of globbing to print out all the Perl scripts one after the other. You will need to use the `open()` function to read from each file in turn. (Answer: `exercises/answers/printscripts.pl`)

Finding information about files

The file test operators are documented fully in **perldoc -f -x**.

We can find out various information about files by using file test operators and functions such as `stat()`.

Table C-1. File test operators

Operator	**Meaning**
`-e`	File exists.
`-r`	File is readable
`-w`	File is writable
`-x`	File is executable
`-o`	File is owned by you
`-z`	File has zero size.
`-s`	File has nonzero size (returns size).
`-f`	File is a plain file (as opposed to a directory, symbolic link, device etc.)
`-d`	File is a directory.
`-l`	File is a symbolic link.
`-p`	File is a named pipe (FIFO), or Filehandle is a pipe.
`-S`	File is a socket.
`-b`	File is a block special file.
`-c`	File is a character special file.
`-t`	Filehandle is opened to a tty.
`-u`	File has setuid bit set.
`-g`	File has setgid bit set.
`-k`	File has sticky bit set.
`-T`	File is a text file (heuristic guess).
`-B`	File is a binary file (opposite of -T).
`-M`	Days since file last modified, when script started.
`-A`	Same for access time.
`-C`	Same for inode change time.

Here’s how the file test operators are usually used:

```
if(not -e "config.txt") {
        die "Config file doesn’t exist";
}
```

The `stat()` function returns similar information for a single file, in list form. `lstat()` can also be used for finding information about a file which is pointed to by a symbolic link. If you’ve used these functions in C or other languages, then you’ll probably find them somewhat familiar in Perl. Check out **perldoc -f stat** to see the format this data is returned in and how to make use of it.

The file test operators expect the file you're testing to be in the current working directory. If this is not the case, make sure you prepend a path to the file before doing your test.

Multiple file tests

Occasionally it is desirable to perform several tests on the same file at the same time. Perhaps you'd like to check that a file is both readable and writable. It is possible to perform your test like this:

```
if (-r $file && -w $file) {
        # ...
}
```

but that involves two separate tests which both take time. The file might also change between the tests (which is why file tests are almost always a bad idea in security situations).

Perl caches the result of file tests in a special filehandle called _ (underscore). Performing tests on this filehandle can often avoid subsequent system calls, resulting in a slight performance gain.

```
if(-r $file && -w _) {
        # ...
}
```

There are some caveats on when the _ filehandle can be used with certain operators such as `-l` and `-t`. To find out more about these and to learn more about file test operators read **perldoc -f -x**.

Exercises

1. Use the file test operators to print out only files from a directory which are "normal" files, i.e. not directories, symbolic links or other oddities. (Answer: `exercises/answers/normaldirlist.pl`)
2. Write a script to find zero-byte files in a directory. (Answer: `exercises/answers/zerobyte.pl`)
3. Write a script to find the largest file in a directory: `exercises/answers/largestfile.pl`)
4. Write a script which asks a user for a file to open, takes their input from STDIN, checks that the file exists, then prints out the contents of that file. (Answer: `exercises/answers/fileexists.pl`)

Changing the working directory

The function `chdir` allows you to change your program's working directory. All relative file access from that point on will use the new directory. Note that this does *not* change the working directory of the calling process.

```
use autodie qw(:all);   # Enables autodie for everything, including system()

# Archive log files
my $tar = "/bin/tar";
my $date = "2007-01-01";
my $directory = "/var/log/apache/";

chdir($directory);

# Get all files in this directory
my @files = glob("*.log.*");
system("$tar -czf weblogs.$date.tgz @files") if @files;
```

You can find more about `chdir` by reading **perldoc -f chdir** or page 688 (page 148, 2nd Ed) in the Camel book.

To find out what your current working directory is, we can use the `Cwd` module:

```
use Cwd;
my $current_working_directory = getcwd();
```

Recursing down directories

The **File::Find** module is documented on pages 889-890 (page 439, 2nd Ed) or more fully in **perldoc File::Find**.

The built-in functions described above do not enable you to easily recurse through subdirectories. Luckily, the **File::Find** module is part of the standard library distributed with Perl 5.

`File::Find` emulates Unix's **find** command. It takes as its arguments a subroutine to execute for each file found, and a list of directories to search. Note that to pass a reference to a subroutine we prefix the name of the subroutine with `\&`. In our example below, this is `\&wanted`.

```
#!/usr/bin/perl
use strict;
use warnings;
use File::Find;

print "Enter the directory to start searching in: ";
chomp(my $dir = <STDIN>);

# find takes a subroutine reference and the directory to start working from.
```

```
find (\&wanted, $dir);

sub wanted {
        if(/\.pl$/) {                          # See if it’s a .pl file
                print "$File::Find::name\n";   # Print the current file name.
        }
}
```

For each file found, certain variables are set.

- `$_` is set to the name of the current file.
- `$File::Find::dir` is set to the directory that contains the file.
- `$File::Find::name` contains the full name of the file, i.e. `$File::Find::dir/$_`.

`File::Find` automatically changes your current working directory to the same as the file you are currently examining. Thus there’s rarely a need to use `$File::Find::dir`. If all you want to do is process the file regardless of its location on the file system you can simply open the file using the name in `$_`. This behaviour can be turned off if desired, see **perldoc File::Find** for further information.

File::Find::Rule

Some people find the call-back interface to `File::Find` difficult to understand. Furthermore, storing both your rules and your actions in the call-back subroutine hides a lot of detail from someone glancing over your code. As a result, an alternative exists called `File::Find::Rule`.

The below traverses the directory trees from `@ARGV` and returns the filenames for files ending in `.mp3` or `.ogg` which are greater than 100Kb and haven’t been accessed for a year or more. We can then work with that list as we see fit.

```
use File::Find::Rule;
my $YEAR_AGO = time() - 365 * 24 * 60 * 60;     # Year ago in seconds
my $SIZE = 100_000;                             # 100k bytes

my @old_music = File::Find::Rule->file()
                                ->name ( ’*.mp3’, ’*.ogg’)
                                ->atime( "< $YEAR_AGO" )
                                ->size ( "> $SIZE" )
                                ->in   ( @ARGV );

# Do something with @old_music files
```

You can read more about `File::Find::Rule` on CPAN (http://search.cpan.org/perldoc?File::Find::Rule).

Exercises

Use either `File::Find` or `File::Find::Rule` for the following exercises.

1. Write a program which print outs the names of plain text files only (hint: use file test operators). A `File::Find` starter can be found in `exercises/find.pl` while a `File::Find::Rule` starter can be found in `exercises/findrule.pl`.
2. Now use it to print out the contents of each text file. You'll probably want to pipe your output through **more** (or your favourite pager) so that you can see it all. (Answer: `exercises/answers/find.pl`)

opendir and readdir

`opendir()` is documented on page 755 (page 195, 2nd Ed) of the Camel book. `readdir()` is on page 770 (page 202, 2nd Ed). Also read **perldoc -f opendir** and **perldoc -f readdir**.

If we are dealing with directories containing many files, or many subdirectories, we may come up against some limitations in `glob()`. Just as it can be more effective to open a file and walk over it line by line; sometimes it's more efficient to open our directory and walk through it file by file without processing those files as `glob()` does.

To open directories for access we can use the `opendir()` function. Once a directory is open, we can read file names from it using the `readdir()` function.

To read the contents of files in the directory, you still need to open each one using the `open()` function.

```
# $ENV{HOME} stores the home directory on Unix platforms, use
# $ENV{HOMEPATH} for MS Windows
my $home = $ENV{HOME} || $ENV{HOMEPATH};

opendir(my $home_dir, $home) or die "Can’t read dir $home: $!";

while(my $file = readdir $home_dir) {
        # Skip over directories and non-plain files (eg devices)
        next unless -f "$home/$file";

        open(my $fh, "<", "$home/$file") or die "Can’t open file $home/$file: $!";

        # Do something with $fh

        close my $fh;
}
closedir $home_dir;
```

The `$home_dir` in the previous example is a *directory handle* not a filehandle. Attempting to use a directory handle as a filehandle (or the opposite) will result in an error.

Package directory handles

Under legacy code you may find package directory handles instead of scalar filehandles. These have all the same problems as package filehandles, while being sufficiently rare that people will accidentally assume they are filehandles anyway.

```
# Package directory handles are a bad idea, but look like this:

opendir(HOMEDIR, $home) or die "Can’t read dir $home: $!";
while(my $file = readdir(HOMEDIR)) {
        # do something with $file
}
```

If you are using Perl 5.6.1 or later, it is recommended that you only use scalar directory handles.

Exercises

1. Use `opendir()` and `readdir()` to obtain a list of files in a directory. What order are they in?
2. Use the `sort()` function to sort the list of files asciibetically (Answer: `exercises/answers/dirlist.pl`)

glob and readdir

There are some major differences between `glob()` and `readdir()`. `glob()` is not as fast but gives you flexibility over which filenames you get back: `glob("*.c")` for example, returns only files with the ".c" extension. `glob()` also gives you back filenames in asciibetical order, whereas `readdir` gives you back the files in whatever order they’re stored in the internal representation of your system.

`glob("some/path/*")` will return filenames with path intact whereas readdir will return only the filenames of the files in the directory.

The last difference between these is their behaviour with "`.`" files. For example "`.bashrc`". `glob("*")` will not return these files (although `glob(".*")` will), whereas `readdir()` will always return "`.`" files.

Table C-2. Differences between glob and readdir

glob	readdir
Slower	Faster
Allows you to filter filenames	Gives you all filenames
Returns files in asciibetical order	Returns files in file-system order
Returns filename with path intact	Returns filename only
Does not return dot files when called as `glob("*")` (although `glob("* .*")` does).	Returns all filenames

Chapter summary

- Angle brackets can also be used as a globbing operator if anything other than a filehandle name appears between the angle brackets. In scalar context, returns the next file matching the glob pattern; in list context, returns all remaining matching files.
- File test operators or `stat()` can be used to find information about files.
- The `opendir()`, `readdir()` and `closedir()` functions can be used to open, read from, and close directories.
- The **File::Find** module can be used to recurse down through directories.

Appendix D. More functions

The grep() function

The `grep()` function is used to search a list for elements which match a certain regexp pattern. It takes two arguments - a pattern and a list - and returns a list of the elements which match the pattern.

The `grep()` function is on page 730 (page 178, 2nd Ed) of your Camel book.

```
# trivially check for valid email addresses
my @valid_email_addresses = grep /\@/, @email_addresses;
```

The `grep()` function temporarily assigns each element of the list to `$_` then performs matches on it.

There are many more complicated uses for the grep function. For instance, instead of a pattern you can supply an entire block which is to be used to process the elements of the list.

```
my @long_words = grep { (length($_) > 8); } @words;

# This is the same as:
my @long_words;
foreach my $word (@words) {
        push @long_words, $word if length $word > 8;
}
```

`grep()` doesn't require a comma between its arguments if you are using a block as the first argument, but does require one if you're just using an expression. Have a look at the documentation for this function to see how this is described.

Exercises

1. Use `grep()` to return a list of elements which contain numbers (Answer: `exercises/answers/grepnumber.pl`)
2. Use `grep()` to return a list of elements which are
    a. keys to a hash (Answer: `exercises/answers/grepkeys.pl`)
    b. readable files (Answer: `exercises/answers/grepfiles.pl`)

The map() function

The `map()` function can be used to perform an action on each member of a list and return the results as a list.

```
my @lowercase = map lc, @words;
my @doubled = map { $_ * 2 } @numbers;
```

`map()` is often a quicker way to achieve what would otherwise be done by iterating through the list with `foreach`.

```
my @lowercase;
foreach (@words) {
        push (@lowercase, lc($_));
}
```

`map` can return multiple arguments for each block evaluation. This can be used to build a bigger list or assign key-value pairs:

```
my %random_scores = map { $_ => rand() } @events;
```

Like `grep()`, it doesn't require a comma between its arguments if you are using a block as the first argument, but does require one if you're just using an expression.

Exercises

1. Create an array of numbers. Use `map()` to calculate the square of each number. Print out the results.

List manipulation with splice()

Earlier we learned that we can use `push`, `pop`, `shift` and `unshift` to add and remove items from either end of an array. However, none of these methods provide an easy way to add and remove items from inside the array. For this we need `splice`. `splice` takes up to four arguments; the array; the offset from the start of the array, the length to replace and the replacement list. It behaves very similiarly to `substr`.

```
my @numbers = (1..5);

# Remove 1 element from offset 2 (removes the number 3)
splice(@numbers, 2, 1);
print "@numbers\n";     # prints 1 2 4 5

# Add (9,9,9) replacing 0 elements from offset 2
splice(@numbers, 2, 0, (9,9,9));
print "@numbers\n";     # prints 1 2 9 9 9 4 5
```

Appendix E. Unix cheat sheet

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

Table E-1. Simple Unix commands

Action	Command
Change to home directory	**cd**
Change to *directory*	**cd *directory***
Change to directory above current directory	**cd ..**
Show current directory	**pwd**
Directory listing	**ls**
Wide directory listing, showing hidden files	**ls -al**
Showing file permissions	**ls -al**
Making a file executable	**chmod +x *filename***
Printing a long file a screenful at a time	**more *filename*** or **less *filename***
Getting help for *command*	**man *command***

Appendix F. Editor cheat sheet

This summary is laid out as follows:

Table F-1. Layout of editor cheat sheets

Running	Recommended command line for starting it.
Using	Really basic howto. This is not even an attempt at a detailed howto.
Exiting	How to quit.
Gotchas	Oddities to watch for.
Help	How to get help.

vi (or vim)

Running

```
% vi filename

      or

% vim filename (where available)
```

Using

- `i` to enter insert mode, then type text, press **ESC** to leave insert mode.
- `x` to delete character below cursor.
- `dd` to delete the current line
- Cursor keys should move the cursor while *not* in insert mode.
- If not, try `hjkl`, `h` = left, `l` = right, `j` = down, `k` = up.
- `/`, then a string, then **ENTER** to search for text.
- `:w` then **ENTER** to save.

Exiting

- Press **ESC** if necessary to leave insert mode.
- `:q` then **ENTER** to exit.
- `:q!` **ENTER** to exit without saving.
- `:wq` to exit with save.

Gotchas

vi has an insert mode and a command mode. Text entry only works in insert mode, and cursor motion only works in command mode. If you get confused about what mode you are in, pressing **ESC** twice is guaranteed to get you back to command mode (from where you press i to insert text, etc).

Help

`:help` **ENTER** might work. If not, then see the manpage.

nano (pico clone)

Running

```
% nano -w filename
```

Using

- Cursor keys should work to move the cursor.
- Type to insert text under the cursor.
- The menu bar has `^X` commands listed. This means hold down **CTRL** and press the letter involved, eg **CTRL-W** to search for text.
- **CTRL-O** to save.

Exiting

Follow the menu bar, if you are in the midst of a command. Use **CTRL-X** from the main menu.

Gotchas

Line wraps are automatically inserted unless the -w flag is given on the command line. This often causes problems when strings are wrapped in the middle of code and similar.

Help

CTRL-G from the main menu, or just read the menu bar.

Appendix G. ASCII Pronunciation Guide

Table G-1. ASCII Pronunciation Guide

Character	Pronunciation
#	hash, pound, sharp, number
!	bang, exclamation
*	star, asterisk
$	dollar
@	at
%	percent, percentage sign
&	ampersand
"	double-quote
’	single-quote, tick, forward tick
( )	open/close parentheses, round brackets, bananas
<	less than
>	greater than
-	dash, hyphen
.	dot
,	comma
/	slash, forward-slash
\	backslash, slosh
:	colon
;	semi-colon
=	equals
?	question-mark
^	caret (pron. carrot), hat
_	underscore
[ ]	open/close bracket, square bracket
{ }	open/close curly brackets, brace
|	pipe, vertical bar, bar
~	tilde, wiggle, squiggle
‘	backtick

Colophon

```
        mJXXLm.                                       .mJXXLm
       JXXXXXXXXL.       JXXLm.       .mJXXL       .JXXXXXXXXL
      {XXXXXXXXXXX.     JXXXmXXXXm mXXXXmXXXL     .XXXXXXXXXXX}
     .XXXXXXXXXXXXXL.  {XXXXXXXXXF 7XXXXXXXXX}  .JXXXXXXXXXXXXX.
    JXXXXXXXXXXXXXXXXL.‘XXXXXX.       .XXXXXX’.JXXXXXXXXXXXXXXXXL
   JXXXXXXXXXXXXXXXXXXXmXXXXXXX.     .XXXXXXXmXXXXXXXXXXXXXXXXXXXL
 .XXXXXXXXXXXXXXXXXXXXXXXXXXXXX}     {XXXXXXXXXXXXXXXXXXXXXXXXXXXXX.
.XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX       XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.
JXXXXXXXXXXXXXXXXXXXXXXXXXXXXXF       7XXXXXXXXXXXXXXXXXXXXXXXXXXXXXL
XX^7XXXXXXXXXXXXXXXXXXXXXXXXXF         7XXXXXXXXXXXXXXXXXXXXXXXXXF^XX
XX {XXXFXXXXXXXXXXXXXXXXXXXF’           ‘7XXXXXXXXXXXXXXXXXXX7XXX} XX
‘X}{XXX’7XXXFXXXXX^XXXXX ‘’               ‘’ XXXXX^XXXXX7XXXF‘XXX}{X’
 ‘’XXX’ {XXX’XXXXX 7XXXF                     7XXXF XXXXX‘XXX} ‘XXX‘’
  .XX}  {XXF {XXXX}‘XXX}                     {XXX’{XXXX} 7XX}  {XX.
  {XX   ‘XXL  ‘7XX} 7XX}                     {XXF {XXF’  JXX’   XX}
  ‘XX    ‘XXL mXXF  {XX                       XX}  7XXm JXX’    XX’
   XX      7XXXF    ‘XX                       XX’    7XXXF      XX
   XX.    JXXXX.     7X.                     .XF     .XXXXL    .XX
  {XXL    7XF7XXX.   {XX                     XX}   .XXXF7XF    JXX}
  ‘XXX’              ‘XXXm                 mXXX’              ‘XXX’
                      ^^^^^               ^^^^^
                   .mJXXLm                  mJXXLm.
   .mJXXL       .JXXXXXXXXL                JXXXXXXXXL.       JXXLm.
mXXXXmXXXL     .XXXXXXXXXXX}              {XXXXXXXXXXX.     JXXXmXXXXm
7XXXXXXXXX}  .JXXXXXXXXXXXXX.            .XXXXXXXXXXXXXL.  {XXXXXXXXXF
   .XXXXXX’.JXXXXXXXXXXXXXXXXL          JXXXXXXXXXXXXXXXXL.‘XXXXXX.
  .XXXXXXXmXXXXXXXXXXXXXXXXXXXL        JXXXXXXXXXXXXXXXXXXXmXXXXXXX.
  {XXXXXXXXXXXXXXXXXXXXXXXXXXXXX.    .XXXXXXXXXXXXXXXXXXXXXXXXXXXXX}
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.  .XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   7XXXXXXXXXXXXXXXXXXXXXXXXXXXXXL  JXXXXXXXXXXXXXXXXXXXXXXXXXXXXXF
    7XXXXXXXXXXXXXXXXXXXXXXXXXF^XX  XX^7XXXXXXXXXXXXXXXXXXXXXXXXXF
     ‘7XXXXXXXXXXXXXXXXXXX7XXX} XX  XX {XXXFXXXXXXXXXXXXXXXXXXXF’
       ‘’ XXXXX^XXXXX7XXXF‘XXX}{X’  ‘X}{XXX’7XXXFXXXXX^XXXXX ‘’
          7XXXF XXXXX‘XXX} ‘XXX‘’    ‘’XXX’ {XXX’XXXXX 7XXXF
          {XXX’{XXXX} 7XX}  {XX.      .XX}  {XXF {XXXX}‘XXX}
          {XXF {XXF’  JXX’   XX}      {XX   ‘XXL  ‘7XX} 7XX}
           XX}  7XXm JXX’    XX’      ‘XX    ‘XXL mXXF  {XX
           XX’    7XXXF      XX        XX      7XXXF    ‘XX
          .XF     .XXXXL    .XX        XX.    JXXXX.     7X.
          XX}   .XXXF7XF    JXX}      {XXL    7XF7XXX.   {XX
        mXXX’              ‘XXX’      ‘XXX’              ‘XXXm
       ^^^^^                                              ^^^^^
```

`The use of a camel image in association with Perl is a trademark of O’Reilly & Associates, Inc. Used with permission.`

The `camel code` that makes up the cover art was written by Stephen B. Jenkins (aka Erudil). When executed, it generates the images of four smaller camels as shown above. A discussion of the camel code in its native habitat can be found on PerlMonks (http://www.perlmonks.org/index.pl?node=camel+code). More information about Stephen B. Jenkins and his work can be found on his website (http://www.Erudil.com).