PerlClass.com's

Perl Training Materials

Christopher Hicks
and
Kirrily Robert

Perl Training Materials
by Christopher Hicks

Open Publications License 1.0

Table of Contents

Chapter 1: Introduction

This chapter will...

Welcome to PerlClass.com's Introduction to Perl training module. This is a training course in which you will learn how to program in the Perl programming language.

1.1 Assumed knowledge

To gain the most from this course, you should:

- Be able to use the Unix operating system
  - Move around the file system
  - Create and edit files
  - Run programs
- Have programmed in least one other language and
  - Understand variables, including data types and arrays
  - Understand conditional and looping constructs
  - Understand the use of subroutines and/or functions
- Basic database theory - tables, records, fields
- Basic HTML - paragraphs, headings, ordered and unordered lists, anchor tags, images, etc.

If you need help with editing files under Unix, a cheat-sheet is available in Appendix A and an editor command summary in Appendix B.

The Unix operating system commands you will need are mentioned and explained very briefly throughout the course - please feel free to ask if you need more help. The required Perl knowledge was covered in PerlClass.com's "Introduction to Perl" course, which many of you will have attended recently. Lastly, an HTML cheat-sheet is provided in Appendix D for those who need reminding.

1.2 Day 1 rough outline

- What is Perl? (30 minutes)
- Creating and running a Perl program (45 minutes)
- *Morning tea* (15 minutes)
- Variable types (45 minutes)
- Operators and Functions (60 minutes)
- *Lunch break* (60 minutes)
- Conditional constructs (45 minutes)
- Subroutines (30 minutes)
- *Afternoon tea* (15 minutes)
- Regular expressions (45 minutes)
- Practical exercises (until finish)

1.3 Day 1 objectives

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

1.4 Day 2 outline

- Revise introduction to Perl material
- File I/O
  - Line input and globbing operators
  - Opening files, directories, and pipes
  - Finding information about files
  - Recursing down directories
  - File locking
  - Handling binary data
- Advanced regular expressions
  - Review of basic regexps
  - Multiline strings
  - Backreferences
- More functions
  - `grep()` and `map()` functions
  - `printf()` and `sprintf()`
  - `pack()` and `unpack()`
  - List manipulation with `splice()`
- System interaction
  - `system()` and `exec()`
  - Backticks
  - Interacting with the file system
  - Dealing with users, groups and permissions
  - Interacting with processes
  - Security considerations
- References and complex data structures
  - Creating and dereferencing
  - Complex data structures
  - Anonymous data structures

1.5 Day 2 objectives

- Be able to open files and directories to read and write data, using various techniques
- Perform tests on files and directories
- Open pipes to read or write data through another program
- Use regular expressions to handle multiline data
- Use backreferences to create complex regular expressions
- Use and understand more complex Perl functions such as `grep()` and `map()`
- Use Perl functions to call system commands
- Use Perl to interact with the file system, users, and processes
- Understand the security implications of running system commands from Perl, and how to increase security
- Understand and use Perl references to create complex data structures and anonymous data structures

1.6 Day 3 outline

- About databases
- Text based ("flat file") databases
- Relational databases
- Tables and relationships
- Structured Query Language (SQL)
- MySQL and other database servers
- Features of MySQL
- Getting MySQL
- Setting up MySQL databases
- The MySQL client
- The DBI and DBD modules
- What is DBI?
- DBI syntax
- DBI exercises
- Extended exercises
- References (optional topic)

1.7 Day 3 objectives

- Understand what a database is and use correct terminology to describe types of databases and parts of databases
- Understand and use flat file or textual databases with Perl
- Understand the advantages and limitations of flat file or textual databases and relational databases
- Understand and use Structured Query Language (SQL) to manipulate data in a relational database
- Know about MySQL and other relational databases suitable for small to medium applications
- Use the MySQL command line client to perform SQL queries
- Understand and use Perl's DBI module to interact with databases
- Use the skills and knowledge learned in this module to create a sample application

1.8 Day 4 outline

- What is CGI? (60 minutes)
- Generating web pages with a Perl script (45 minutes)
- Practical exercises (45 minutes)
- Accepting and processing form input with the CGI module (60 minutes)
- *Lunch break*
- Practical examples (60 minutes)
- Security issues (45 minutes)
- Other related features and Perl modules (60+ minutes)

1.9 Day 4 objectives

- Understand the meaning of CGI and the Hypertext Transfer Protocol
- Know how to generate simple web pages using Perl
- Understand how to accept and process data from web forms using the CGI module
- Understand security issues pertaining to CGI programming and how to avoid security problems
- Recognise and use a number of Perl modules for purposes related to CGI programming

1.10 Other topics we can discuss

- Win32 – Perl programming in Windows
- XML – there seems to be a lot of XML data lately
- Tk – GUI toolkit
- mod_perl – Perl integration with apache
- Inline – seamless inclusion of non-Perl in Perl
- Data::Dumper – a compvenient way to print out complex data structures
- DBIx::Class – a friendy oop-style layer on top of DBI
- Storable – persistance of complex Perl object across processes, systems, etc.
- ???
- ???
- ???

1.11 Platform and version details

This course is taught using Linux, a Unix-like operating system. Most of what is learned will work equally well on Microsoft Windows, MacOS or other operating systems. Your instructor will inform you throughout the course of any areas which differ.

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

1.12 The course notes

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

The following typographical conventions are used in these notes:

System commands appear in **this typeface**

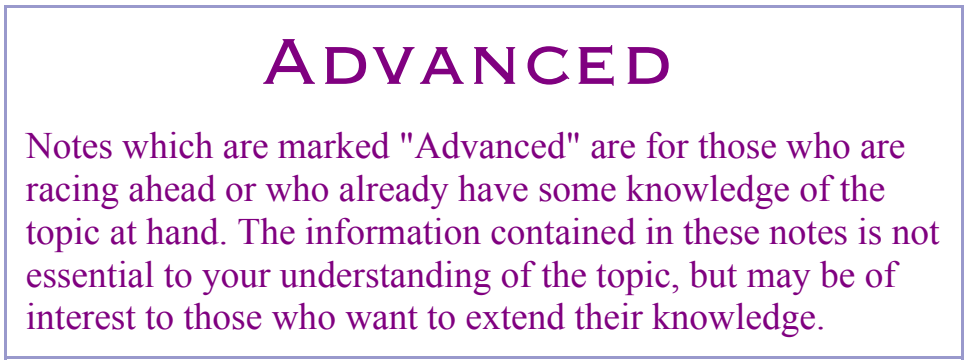
Literal text which you should type in to the command line or editor appears as `monospaced font`.

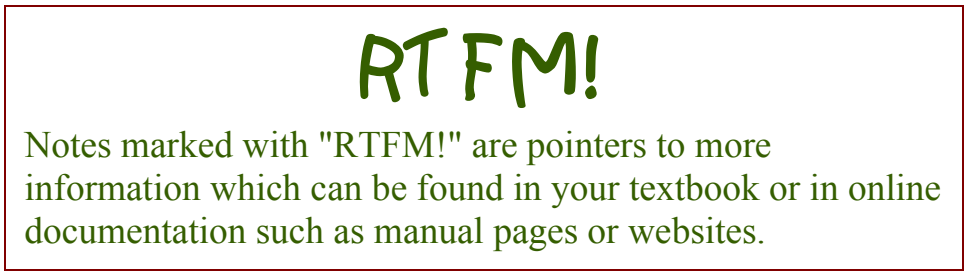
Keystrokes which you should type appear like this: **ENTER**. Combinations of keys appear like this: **CTRL-D**

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

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

Notes and tips appear offset from the text like this.





RTFM!

Src	Chap	Pgs	#
Nutshell 2nd			*Perl in a Nutshell*
Camel 2nd			*Programming Perl*
Camel 3rd			*Programming Perl*
perldoc			perldoc online
Cookbook 2nd			*Perl Cookbook*
Learning 3rd			*Learning Perl*
Learning 4th			*Learning Perl*

Most RTFM boxes will appear with a table like this. The "src" column refers to a variety of standard Perl references. "Chap" is the chapter which for electronic contexts like man and perldoc would refer to the "man page" or "whole pod".

1.13 Other materials

In addition to these notes, you should have a copy of the required text book for this course: *Perl in a Nutshell* 2nd Ed. by Nathan Patwardhan, Ellen Siever and Stephen Spainhour. The Nutshell will be used throughout the course, and will be a valuable reference to take home and keep next to your computer.

Chapter 2: What is Perl

In this chapter...

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

2.1 Perl's name

Perl has been said to stand for "Practical Extraction and Reporting Language" (by it's fans) or "Pathologically Eclectic Rubbish Lister" (by its detractors). In fact, Perl is not an acronym; it's a shortened version of the program's original name, "pearl", and when you're talking 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 chainsaw 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.

2.2 Typical uses of Perl

2.2.1 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.

2.2.2 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.

2.2.3 CGI and web programming

Since HTML is just text with built-in formatting, Perl can be used to process and generate HTML. Perl is probably the most popular language around for web development, and there are many tools and scripts available for free.

2.2.4 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.

2.2.5 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.

2.2.6 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, pron. "piddle") makes number-crunching easy, and there is even a Perl Object Environment (POE) which is used for event-driven state machines.

2.3 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 a decade of optimizations 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 pageful 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.

2.4 The Perl Philosophy

2.4.1 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 them *wrong*.

2.4.2 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.

Of course, some Perl programs are more correct than others, but while elegance is a fine thing to strive for, most Perl people realise that soemtimes you just have to write a quick and dirty hack that'll keep things running for the mean time. If you get the time to make it beautiful later, so much the better.

2.4.3 Three virtues of a programmer

The Camel book contains the following entries in its glossary:

2.4.3.1 Laziness

The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-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.

2.4.3.2 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.

2.4.3.3 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.

2.4.4 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.

2.4.5 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) helps give them the financial means to keep supporting Perl.

2.5 Parts of Perl

2.5.1 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 5.005, which is available from the Perl website (http://www.perl.com/) or any of a number of mirror sites (the Windows version is available from ActiveState (http://www.activestate.com/). The next release of Perl will be version 5.6; the jump in version numbers is because it was felt that the number of additional features between releases warranted a larger difference between version numbers.

2.5.2 Manuals

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**

Watch while this is demonstrated; you'll get a chance to try it soon.

2.5.3 Perl Modules

Perl also comes with a collection of modules. These are Perl programs 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.perl.com/CPAN)) and installed separately.

2.6 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
  - library modules

Chapter 3: 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.

3.1 Logging into your account

Your username and password will have been given to you with these course notes.

Table 3-1. Details required to connect to the PerlClass.com training server

Hostname or IP address	
Your username	
Your password	

1. Open putty
2. Connect to the training server at the hostname or IP number given above
3. Login using the username and password you were given

You will find yourself at a Unix shell prompt. Hopefully (if you met the pre-requisites of this course) you will now be able to see that your account has a subdirectory called `exercises/` which are the example scripts and exercises given in these course notes. If you're not quite up to speed with Unix, there's a cheat-sheet in Appendix A of these notes.

3.2 Using perldoc

On the command line, type **perldoc perl**. You will find yourself in the Perl documentation pages. Here's how to get around inside the documentation:

Table 3-2. Getting around in perldoc

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

```
$ perldoc perl
PERL(1)            User Contributed Perl Documentation              PERL(1)

NAME
       perl - Practical Extraction and Report Language

SYNOPSIS
       perl [ -sTuU ] [ -hv ] [ -V[:configvar] ]
           [ -cw ] [ -d[:debugger] ] [ -D[number/list] ]
           [ -pna ] [ -Fpattern ] [ -l[octal] ] [ -0[octal] ]
           [ -Idir ] [ -m[-]module ] [ -M[-]'module...' ]
           [ -P ] [ -S ] [ -x[dir] ]
           [ -i[extension] ] [ -e 'command' ]
           [ -- ] [ program-file ] [ argument ]...

       If you're new to Perl, you should start with perlintro, which is a
       general intro for beginners and provides some background to help
       you navigate the rest of Perl's extensive documentation.

       For ease of access, the Perl manual has been split up into several
       sections.

       Overview

           perl                Perl overview (this section)
           perlintro           Perl introduction for beginners
           perltoc             Perl documentation table of contents
```

Tutorials

perlreftut	Perl references short introduction
perldsc	Perl data structures intro
perllol	Perl data structures: arrays of arrays
perlrequick	Perl regular expressions quick start
perlretut	Perl regular expressions tutorial
perlboot	Perl OO tutorial for beginners
perltoot	Perl OO tutorial, part 1
perltooc	Perl OO tutorial, part 2
perlbot	Perl OO tricks and examples
perlstyle	Perl style guide
perlcheat	Perl cheat sheet
perltrap	Perl traps for the unwary
perldebtut	Perl debugging tutorial
perlfaq	Perl frequently asked questions
perlfaq1	General Questions About Perl
perlfaq2	Obtaining and Learning about Perl
perlfaq3	Programming Tools
perlfaq4	Data Manipulation
perlfaq5	Files and Formats
perlfaq6	Regexes
perlfaq7	Perl Language Issues
perlfaq8	System Interaction
perlfaq9	Networking

Reference Manual

perlsyn	Perl syntax
perldata	Perl data structures
perlop	Perl operators and precedence
perlsub	Perl subroutines
perlfunc	Perl built-in functions
perlopentut	Perl open() tutorial
perlpacktut	Perl pack() and unpack() tutorial
perlpod	Perl plain old documentation

perlpodspec	Perl plain old documentation format specification
perlrun	Perl execution and options
perldiag	Perl diagnostic messages
perllexwarn	Perl warnings and their control
perldebug	Perl debugging
perlvar	Perl predefined variables
perlre	Perl regular expressions, the rest of the story
perlreref	Perl regular expressions quick reference
perlref	Perl references, the rest of the story
perlform	Perl formats
perlobj	Perl objects
perltie	Perl objects hidden behind simple variables
perldbmfilter	Perl DBM filters
perlipc	Perl interprocess communication
perlfork	Perl fork() information
perlnumber	Perl number semantics
perlthrtut	Perl threads tutorial
perlothrtut	Old Perl threads tutorial
perlport	Perl portability guide
perllocale	Perl locale support
perluniintro	Perl Unicode introduction
perlunicode	Perl Unicode support
perlebcdic	Considerations for running Perl on EBCDIC platforms
perlsec	Perl security
perlmod	Perl modules: how they work
perlmodlib	Perl modules: how to write and use
perlmodstyle	Perl modules: how to write modules with style
perlmodinstall	Perl modules: how to install from CPAN
perlnewmod	Perl modules: preparing a new module for distribution
perlutil	utilities packaged with the Perl

distribution

perlcompile Perl compiler suite intro

perlfilter Perl source filters

Internals and C Language Interface

perlembed Perl ways to embed perl in your C or C++ application
perldebguts Perl debugging guts and tips
perlxstut Perl XS tutorial
perlxs Perl XS application programming interface
perlclib Internal replacements for standard C library functions
perlguts Perl internal functions for those doing extensions
perlcall Perl calling conventions from C

perlapi Perl API listing (autogenerated)
perlintern Perl internal functions (autogenerated)
perliol C API for Perl's implementation of IO in Layers
perlapio Perl internal IO abstraction interface

perlhack Perl hackers guide

Miscellaneous

perlbook Perl book information
perltodo Perl things to do

perldoc Look up Perl documentation in Pod format

perlhist Perl history records
perldelta Perl changes since previous version
perl584delta Perl changes in version 5.8.4
perl583delta Perl changes in version 5.8.3
perl582delta Perl changes in version 5.8.2
perl581delta Perl changes in version 5.8.1

perl58delta Perl changes in version 5.8.0
perl573delta Perl changes in version 5.7.3
perl572delta Perl changes in version 5.7.2
perl571delta Perl changes in version 5.7.1
perl570delta Perl changes in version 5.7.0
perl561delta Perl changes in version 5.6.1
perl56delta Perl changes in version 5.6
perl5005delta Perl changes in version 5.005
perl5004delta Perl changes in version 5.004

perlartistic Perl Artistic License
perlgpl GNU General Public License

Language-Specific

perlcn Perl for Simplified Chinese (in EUC-CN)
perljp Perl for Japanese (in EUC-JP)
perlko Perl for Korean (in EUC-KR)
perltw Perl for Traditional Chinese (in Big5)

Platform-Specific

perlaix Perl notes for AIX
perlamiga Perl notes for AmigaOS
perlapollo Perl notes for Apollo DomainOS
perlbeos Perl notes for BeOS
perlbs2000 Perl notes for POSIX-BC BS2000
perlce Perl notes for WinCE
perlcygwin Perl notes for Cygwin
perldgux Perl notes for DG/UX
perldos Perl notes for DOS
perlepoc Perl notes for EPOC
perlfreebsd Perl notes for FreeBSD
perlhpux Perl notes for HP-UX
perlhurd Perl notes for Hurd
perlirix Perl notes for Irix
perlmachten Perl notes for Power MachTen
perlmacos Perl notes for Mac OS (Classic)
perlmacosx Perl notes for Mac OS X
perlmint Perl notes for MiNT

```
perlmpeix           Perl notes for MPE/iX
perlnetware         Perl notes for NetWare
perlos2             Perl notes for OS/2
perlos390           Perl notes for OS/390
perlos400           Perl notes for OS/400
perlplan9           Perl notes for Plan 9
perlqnx             Perl notes for QNX
perlsolaris         Perl notes for Solaris
perltru64           Perl notes for Tru64
perluts             Perl notes for UTS
perlvmesa           Perl notes for VM/ESA
perlvms             Perl notes for VMS
perlvos             Perl notes for Stratus VOS
perlwin32           Perl notes for Windows
```

By default, the manpages listed above are installed in the /usr/local/man/ directory.

Extensive additional documentation for Perl modules is available. The default configuration for perl will place this additional documentation in the /usr/local/lib/perl5/man directory (or else in the man subdirectory of the Perl library directory). Some of this additional documentation is distributed standard with Perl, but you'll also find documentation for third-party modules there.

You should be able to view Perl's documentation with your man(1) program by including the proper directories in the appropriate start-up files, or in the MANPATH environment variable. To find out where the configuration has installed the manpages, type:

```
perl -V:man.dir
```

If the directories have a common stem, such as /usr/local/man/man1 and /usr/local/man/man3, you need only to add that stem (/usr/local/man) to your man(1) configuration files or your MANPATH environment variable. If they do not share a stem, you'll have to add both stems.

If that doesn't work for some reason, you can still use the supplied perldoc script to view module information. You might also look into getting a replacement man program.

If something strange has gone wrong with your program and you're not sure where you should look for help, try the -w switch first. It will often point out exactly where the trouble is.

DESCRIPTION

Perl is a language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal).

Perl combines (in the author's opinion, anyway) some of the best features of C, **sed**, **awk**, and **sh**, so people familiar with those languages should have little difficulty with it. (Language historians will also note some vestiges of **csh**, Pascal, and even BASIC-PLUS.) Expression syntax corresponds closely to C expression syntax. Unlike most Unix utilities, Perl does not arbitrarily limit the size of your data—if you've got the memory, Perl can slurp in your whole file as a single string. Recursion is of unlimited depth. And the tables used by hashes (sometimes called "associative arrays") grow as necessary to prevent degraded performance. Perl can use sophisticated pattern matching techniques to scan large amounts of data quickly. Although optimized for scanning text, Perl can also deal with binary data, and can make dbm files look like hashes. Setuid Perl scripts are safer than C programs through a dataflow tracing mechanism that prevents many stupid security holes.

If you have a problem that would ordinarily use **sed** or **awk** or **sh**, but it exceeds their capabilities or must run a little faster, and you don't want to write the silly thing in C, then Perl may be for you. There are also translators to turn your **sed** and **awk** scripts into Perl scripts.

But wait, there's more...

Begun in 1993 (see perlhist), Perl version 5 is nearly a complete rewrite that provides the following additional benefits:

- modularity and reusability using innumerable modules

  Described in perlmod, perlmodlib, and perlmodinstall.

- embeddable and extensible

  Described in perlembed, perlxstut, perlxs, perlcall, perlguts, and xsubpp.

- roll-your-own magic variables (including multiple simultaneous DBM implementations)

  Described in perltie and AnyDBM_File.

- subroutines can now be overridden, autoloaded, and prototyped

  Described in perlsub.

- arbitrarily nested data structures and anonymous functions

  Described in perlreftut, perlref, perldsc, and perllol.

- object-oriented programming

  Described in perlobj, perlboot, perltoot, perltooc, and perlbot.

- support for light-weight processes (threads)

  Described in perlthrtut and threads.

- support for Unicode, internationalization, and localization

  Described in perluniintro, perllocale and Locale::Maketext.

- lexical scoping

  Described in perlsub.

- regular expression enhancements

  Described in perlre, with additional examples in perlop.

- enhanced debugger and interactive Perl environment, with integrated editor support

Described in perldebtut, perldebug and perldebguts.

- POSIX 1003.1 compliant library

  Described in POSIX.

Okay, that's definitely enough hype.

AVAILABILITY

Perl is available for most operating systems, including virtually all Unix-like platforms. See "Supported Platforms" in perlport for a listing.

ENVIRONMENT

See perlrun.

AUTHOR

Larry Wall <larry@wall.org>, with the help of oodles of other folks.

If your Perl success stories and testimonials may be of help to others who wish to advocate the use of Perl in their applications, or if you wish to simply express your gratitude to Larry and the Perl developers, please write to perl-thanks@perl.org .

FILES

"@INC"	locations of perl libraries

SEE ALSO

a2p	awk to perl translator
s2p	sed to perl translator
http://www.perl.com/	the Perl Home Page
http://www.cpan.org/	the Comprehensive Perl Archive
http://www.perl.org/	Perl Mongers (Perl user groups)

DIAGNOSTICS

The "use warnings" pragma (and the **-w** switch) produces some lovely diagnostics.

See perldiag for explanations of all Perl's diagnostics. The "use

diagnostics" pragma automatically turns Perl's normally terse warnings and errors into these longer forms.

Compilation errors will tell you the line number of the error, with an indication of the next token or token type that was to be examined. (In a script passed to Perl via **-e** switches, each **-e** is counted as one line.)

Setuid scripts have additional constraints that can produce error messages such as "Insecure dependency". See perlsec.

Did we mention that you should definitely consider using the **-w** switch?

BUGS

The -w switch is not mandatory.

Perl is at the mercy of your machine's definitions of various operations such as type casting, atof(), and floating-point output with sprintf().

If your stdio requires a seek or eof between reads and writes on a particular stream, so does Perl. (This doesn't apply to sysread() and syswrite().)

While none of the built-in data types have any arbitrary size limits (apart from memory size), there are still a few arbitrary limits: a given variable name may not be longer than 251 characters. Line numbers displayed by diagnostics are internally stored as short integers, so they are limited to a maximum of 65535 (higher numbers usually being affected by wraparound).

You may mail your bug reports (be sure to include full configuration information as output by the myconfig program in the perl source tree, or by "perl -V") to perlbug@perl.org . If you've succeeded in compiling perl, the **perlbug** script in the utils/ subdirectory can be used to help mail in a bug report.

Perl actually stands for Pathologically Eclectic Rubbish Lister, but don't tell anyone I said that.

NOTES

```
       The Perl motto is "There's more than one way to do it."  Divining
       how many more is left as an exercise to the reader.

       The three principal virtues of a programmer are Laziness,
       Impatience, and Hubris.  See the Camel Book for why.

perl v5.8.5                       2005-12-21                          PERL(1)
```

As you can see, there is a lot of documentation included with Perl.

3.3 Using the editor

A Perl script is just a normal text file, which means that you can edit it using a normal text editor.

The system you are using has several editors available for your use, including **vi**, **pico**, and others. Those who are not already familiar with **vi** should probably use **pico**, as it has a simpler interface. If you're an **emacs** user, sorry, our server doesn't have the resources to run it, but we do have other editors which have an **emacs** emulation mode.

To edit a file using pico, type:

```
% pico filename
```

(Note that the percent sign is your unix command line prompt - you don't have to type it.)

To edit a file using vi, type:

```
% vi filename
```

For other editors, just type the name of the editor followed by the name of the file you wish to edit.

A summary of editor commands appears in Appendix B in the back of these course notes, just in case you need them.

Incidentally, Appendix C contains a guide to pronouncing ASCII characters, especially punctuation. This will help you translate perl into spoken language, for ease of communication with other programmers.

3.4 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 later today)
- Each statement ends in a semi-colon
- Statements can flow over several lines
- Whitespace (spaces, tabs and newlines) are ignored most places in a Perl script.

Now, just for practice, open a file called `hello.pl` in your text 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.

3.5 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.

3.6 The "shebang" line

So what if we want to run our program from the command line without having to type in the name of the Perl interpreter first?

You can make a file executable by typing:

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

at the command line. (For more information about the **chmod** command, type **man chmod**).

In order to let the shell know what to do with our program when we try to run it with **./hello.pl** from the command line, we put the following line at the top of our program:

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

That's what we call a "shebang" line (because the `#` is a "hash" sign, and the `!` is referred to as a "bang", hence "hashbang" or "shebang"). It tells the system what to use to interpret our script. Of course, if the Perl interpreter were somewhere else on our system, we'd have to change the shebang line to reflect that.

3.7 Comments

Incidentally, 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.

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

3.8 Command line options

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

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

(Incidentally, it's always a good idea to turn on warnings while you're developing something.)

ADVANCED

Setting the special variable `$^W` to a true value will locally disable warnings (i.e. in the current block).

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd	3	35-38	
Camel 2nd	6	330-337	"Switches"
Camel 3rd	19	486-505	
perldoc	perlrun		
Cookbook 2nd			
Learning 3rd	2	26-27	
Learning 4th			

3.9 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 4: Perl variables

In this chapter...

In this section 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.

4.1 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".

4.2 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, though - 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 depending on what sort of variable they are:

Table 4-1. 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 it.)

4.3 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 use them. 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, in fact, the case.

4.3.1 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

4.3.2 Arguments against strictness

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

Sometimes a little bit of fascism 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.

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd	8	335-336	
Camel 2nd	7	500	
Camel 3rd	4	137-138	
perldoc	strict		
Cookbook 2nd			
Learning 3rd	B	289	
Learning 4th			

4.4 Using the strict pragma

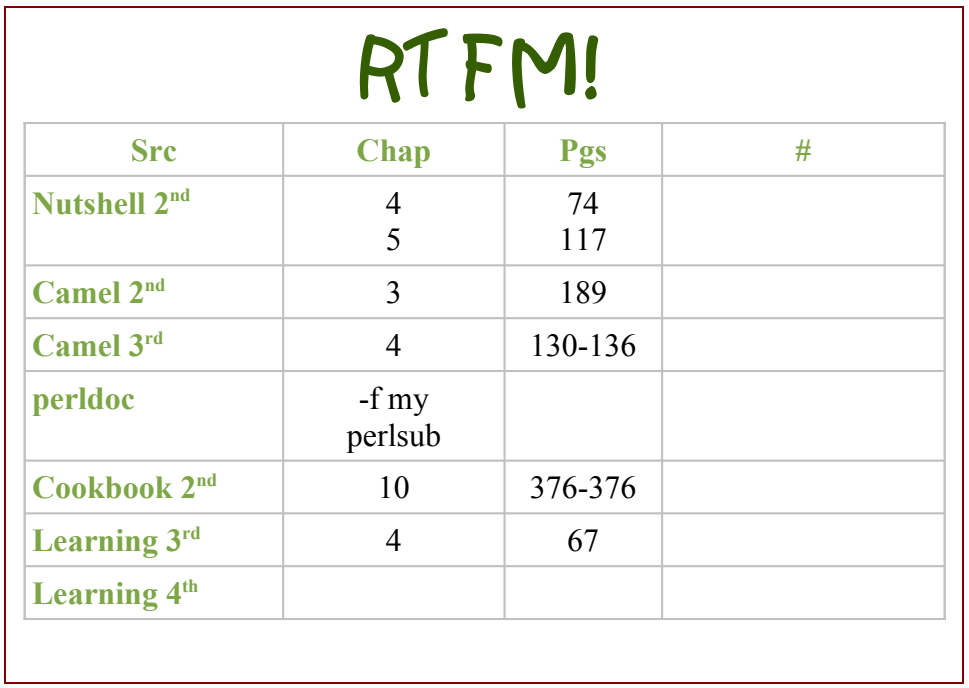
In the interests of bug-free code and teaching better Perl style, we're going to use the strict pragma throughout this training course. Here's how it's invoked:

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

use strict;
```

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

Once we use the strict pragma, we have to explicitly declare new variables using `my`. You'll see this in use below, and it will be discussed again later when we talk about blocks and subroutines.

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.



Src	Chap	Pgs	#
Nutshell 2nd	4 5	74 117	
Camel 2nd	3	189	
Camel 3rd	4	130-136	
perldoc	-f my perlsub		
Cookbook 2nd	10	376-376	
Learning 3rd	4	67	
Learning 4th			

4.5 Scalars

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

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

Here's how we assign values to scalar variables:

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

> ADVANCED
>
> There are other ways to assign things apart from the = operator, too. They're covered on pages 92-93 of the Camel.

As you can see, a scalar can be text of any length, and numbers of any precision (machine dependent, of course). 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 int in the middle of a string we
# want to print
print "$name says, 'The meaning of life is $meaning_of_life.'\n";
```

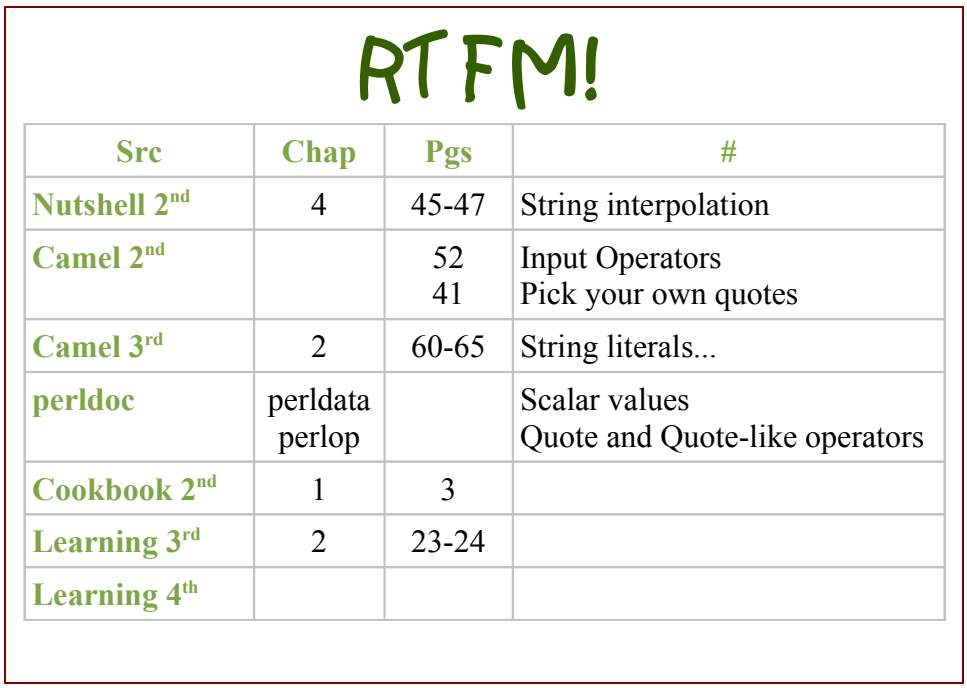
This may seem extraordinarily alien to those used to strictly typed languages,

but believe it or not, the ability to transparently convert between variable types is one of the great strengths of Perl. Some people say that it's also one of the great weaknesses.

Advanced

You can explicitly cast scalars to various specific data types. Look up `int()` on page 180 of the camel, for instance.

4.6 Double and single quotes



Src	Chap	Pgs	#
Nutshell 2nd	4	45-47	String interpolation
Camel 2nd		52 41	Input Operators Pick your own quotes
Camel 3rd	2	60-65	String literals...
perldoc	perldata perlop		Scalar values Quote and Quote-like operators
Cookbook 2nd	1	3	
Learning 3rd	2	23-24	
Learning 4th			

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 last 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;
```

The above example is available in your directory as `exercises/interpolate.pl` so you can experiment with different kinds of quotes.

Note that 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.

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";
```

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: `\\`

4.6.1 Exercises

1. Write a script which sets some variables:
    a. your name
    b. your street number
    c. your favorite colour
2. Print out the values of these variables using double quotes for variable interpolation
3. Change the quotes to single quotes. What happens?
4. Write a script which prints out `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`.

4.7 Arrays

If you think of a scalar as being a singular thing, arrays are the plural form. Just as you have a flock of sheep or a bunch 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), and are initialized thus:

```
my @fruit = ( "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 you've got the memory.

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd	4	47-49	
Camel 2nd	1 2	6 47-49	
Camel 3rd	1 2	8-10 72-76	
perldoc	perldata		
Cookbook 2nd	4	110-149	
Learning 3rd	3	40-55	
Learning 4th			

So if we don't know how many items there are in an array, how can we find out? Well, there are a couple of ways.

First of all, Perl's arrays 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"
```

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 of the Camel: 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.

If we wanted what we call an *array slice* we could say:

```
@fruits[1,2,3];          # oranges, guavas, passionfruit
@magic_numbers[0..1];    # 23, 42
```

You just learned something new, by the way: the `..` ("dot dot") range operator (see pages 90-91 of your Camel or **perldoc perlop**) which creates a temporary list of numbers between the two you specify - in this case 0 and 1, but it could have been 1 and 100 if we'd had an array big enough to use it on. You'll run into this operator again and again, so remember it.

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

```
print "My favorite fruits are @fruits\n";      # whole array
print "Two types of fruit are @fruits[0,2]";    # array slice
```

Returning to the point, how do we find the last element in an array? Well, there's a special variable called `$#array` which is the index of the last element, so you can say:

```
@fruit[0..$#fruit];
```

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

But wait! There's More Than One Way To Do It - and an easier way, at that. If

you evaluate the array in a scalar context - that is, do something like this:

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

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

There's more than two ways to do it, as well - `scalar(@fruits)` and `int(@fruits)` will also tell us how many elements there are in the array.

> **ADVANCED**
>
> Using `$count = scalar @fruits` is the clearest way to express "how many are in fruits?" and is considered a best practice.

4.7.1 A quick look at context

There's a term you've heard used just recently but which hasn't been explained: *context*.

All Perl expressions are evaluated in a context. The two main contexts are:

- scalar context, and
- list context

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

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

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

4.7.2 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
@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.

4.7.3 Exercises

1. Create an array of your friends' names
2. Print out the first element
3. Print out the last element
4. Print out the array from within a double-quoted string using variable interpolation
5. Print out an array slice of the 2nd to 4th items using variable interpolation

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

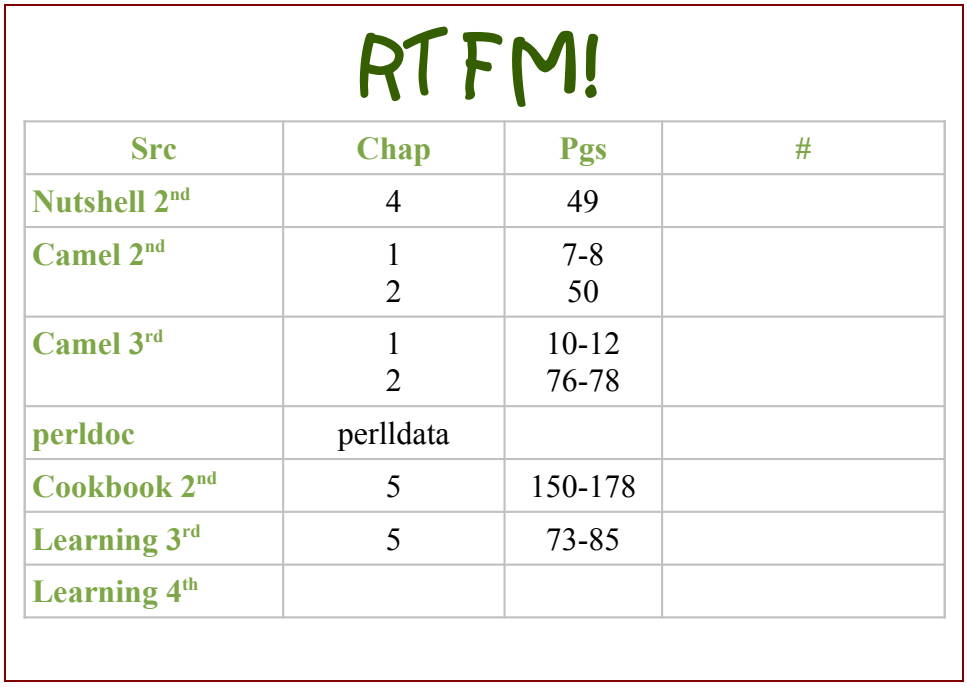
4.7.4 Advanced exercises

1. Print the array without putting quotes around its name. What happens?
2. Set the special variable `$,` to something appropriate and try the previous step again (see page 132 of your Camel for this variable's documentation)
3. What happens if you have a small array and then you assign a value to `$array[1000]`?

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

4.8 Hashes

A hash is a two-dimensional array which contains keys and values. Instead of looking up items in a hash by an array index, you can look up values by their keys.



Src	Chap	Pgs	#
Nutshell 2nd	4	49	
Camel 2nd	1 2	7-8 50	
Camel 3rd	1 2	10-12 76-78	
perldoc	perlldata		
Cookbook 2nd	5	150-178	
Learning 3rd	5	73-85	
Learning 4th			

4.8.1 Initialising a hash

Hashes are initialized 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. Also, you don't need to put quotes on the item which comes immediately before the `=>` operator:

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

4.8.2 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 belonging to January".

4.8.3 Adding new hash elements

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

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

4.8.4 Other things about hashes

- Hashes have no internal order
- There is no equivalent to `$#array` to get the size of a hash
- However, 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.

ADVANCED

You may like to look up the following functions which related to hashes: `keys()`, `values()`, `each()`, `delete()`, `exists()`, and `defined()`.

4.8.5 What's the difference between a hash and an associative array?

Back in the days of Perl version 4 (and earlier), hashes were called associative arrays. The name "hash" is now preferred because it's much quicker to type. If you consider all the times that hashes are talked about in the newsgroup comp.lang.perl.misc (news:comp.lang.perl.misc) and other Perl newsgroups, the renaming of associative arrays to hashes has resulted in a major saving of bandwidth.

4.8.6 Exercises

1. Create a hash of people and something interesting about them
2. Print out a given person's interesting fact
3. Change an 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?

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

4.9 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, or read a special variable to find out the command line arguments passed to your script.

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

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd	4	53-59	
Camel 2nd	2 7	127-140 403	
Camel 3rd	28 32	653-676 884	
perldoc	perlvar English		English provides friendlier names for special variables
Cookbook 2nd			
Learning 3rd	3	49	$_ quickly
Learning 4th			

4.10 The first special variable, $_

The first special variable, and possibly the one 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 can think of Perl variables as being "nouns", then `$_` is the pronoun "it".

4.10.1.1 Exercises

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

The answers to the above exercise are in `exercises/answers/scalars2.pl`.

4.11 @ARGV - a special array

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

```
perl 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`. Try this script, which you'll find in your directory. It's called `exercises/printargs.pl`.

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

print "@ARGV\n";
```

To run the script, type:

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

You should see "foo bar baz" printed out.

4.11.1.1 Exercises

1. Modify your earlier array-printing script to print out the script's command line arguments instead of the names of your friends. Call your script by typing **./scriptname.pl firstarg secondarg thirdarg** or similar.

The answers to the above exercise is in `exercises/answers/argv.pl`

4.12 %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. To view these variables under Unix, simply type **setenv** (C-type shells) or **export** (sh type shells) on the command line.

4.12.1.1 Exercises

1. A user's home directory is stored in the environment variable `HOME`. Print out your own home directory.

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

4.13 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 (`$`) at the beginning of the variable name.
- Scalars can contain strings, numbers, etc
- Strings must be delimited by quote marks. Using double quote marks will allow you to interpolate other variables and meta-characters such as `\n` (newline) into a string. Single quotes do not interpolate.
- Arrays are one-dimensional lists of scalars and are indicated by an at sign (`@`) at the beginning of the variable name.
- Arrays are initialised using a comma-separated list of scalars inside round brackets.
- Arrays are indexed from zero
- Item *n* of an array can be accessed by using `$arrayname[`*n*`]`
- The index of the last item of an array can be accessed by using `$#arrayname`.
- The number of elements in an array can be found by interpreting the array in a scalar context, eg `my $items = @array;`
- Hashes are two-dimensional arrays of keys and values, and are indicated by a percent sign (`%`) at the beginning of the variable name.
- Hashes are initialised using a comma-separated list of scalars inside curly brackets. Whitespace and the `=>` operator (which is syntactically identical to the comma) can be used to make hash assignments look neater.
- The value of a hash item whose key is foo can be accessed by using `$hashname{foo}`
- Hashes have no internal order
- `$_` is a special variable which is the default argument for many Perl functions and operators
- The special array `@ARGV` contains all command line parameters passed to the

script

- The special hash `%ENV` contains information about the user's environment.

Chapter 5: 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.

5.1 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:

5.2 Operators

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd	4	60 - 65	
Camel 2nd	2	76 - 94	
Camel 3rd	3	86 - 110	
perldoc	perlop		
Cookbook 2nd			
Learning 3rd	2	28 - 34	
Learning 4th			

5.3 Arithmetic operators

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

Table 5-1. 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`)

ADVANCED

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
```

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

You can also use `$a++` and `$a----` if you're familiar with such things. `++$a` and `----$a` are also valid, but they do some slighty different things and you won't need them today (but you can read about them on pages 17 to 18 of the Camel if you are sufficiently interested).

5.4 String operators

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

Table 5-2. 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"

5.4.1 Exercises

1. Calculate the cost of 18 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. Use `exercises/operate.pl` to practice using arithmetic and string operators.

5.5 File operators

We can use file test operators to test various attributes of files and directories:

Table 5-3. File test operators

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

5.6 Other operators

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

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

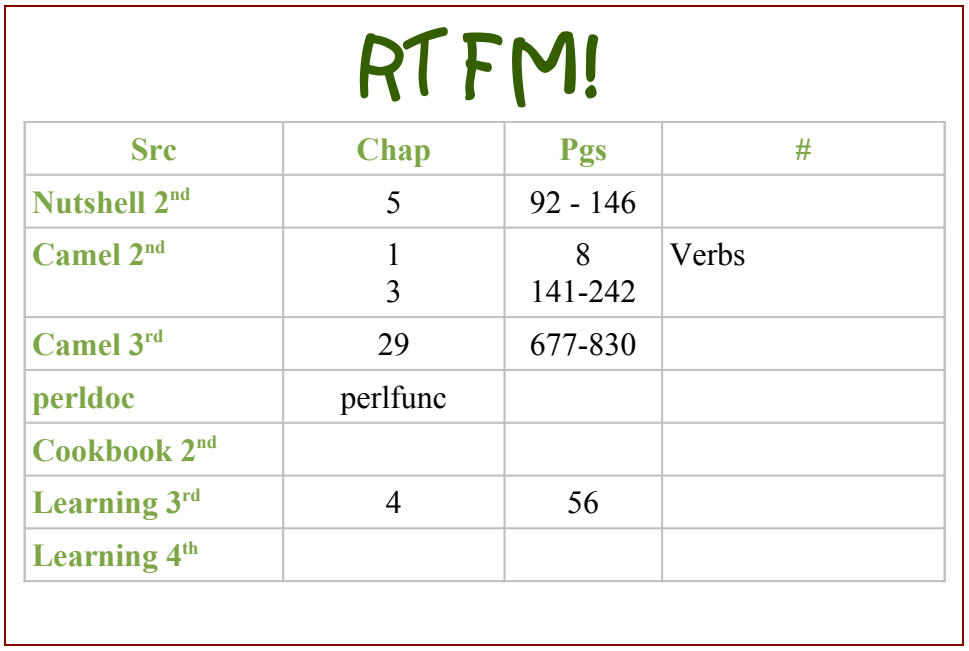
A trinary operator also exists, but we don't deal with it in this course. C programmers will probably already know about it, and can use it if they want.

5.7 Functions

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

- longer names
- can take any kinds 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.



Src	Chap	Pgs	#
Nutshell 2nd	5	92 - 146	
Camel 2nd	1 3	8 141-242	Verbs
Camel 3rd	29	677-830	
perldoc	perlfunc		
Cookbook 2nd			
Learning 3rd	4	56	
Learning 4th			

5.7.1 Types of arguments

Functions typically take the following kind of arguments:

SCALAR -- Any scalar variable - `42`, `"foo"`, or `$a`

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

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

HASH -- Any named or unnamed hash

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

FILEHANDLE -- A filehandle indicating a file that you've opened or one of the

pseudo-files that is automatically opened, such as STDIN, STDOUT, and STDERR

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

5.7.2 Return values

Just as a function can take arguments of various kinds, they can return various things for you to use - though they don't have to, and you don't have to use them if you don't want.

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 could just say:

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

While we're at it, you should also know that the brackets on functions are optional if it's not likely to cause confusion. What's likely to cause confusion varies from one person to the next, but it's a pretty safe bet to use brackets as much as possible when you're starting out, and then drop them off if you see that other people are usually doing it. Seriously. You can learn a lot about Perl style by looking at other people's code, especially code found on CPAN or given as examples in Perl books, newsgroups, etc.

5.8 More about context

Many different functions and operators behave differently depending on whether they're called in *scalar context* or *list context*. Each one will be noted in its documentation, either in the Camel or in the manual pages.

Here are some Perl operators and functions that care about context:

Table 5-4. Context-senstive 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 elements) 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".

You can also use **perldoc -f *`functionname`*** to get the documentation for just a single function.

5.9 Some easy functions

5.9.1 String manipulation

5.9.1.1 Finding the length of a string

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

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

use strict;

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

5.9.1.2 Case conversion

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

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

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

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

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

print lcfirst("Hello, World!");         # prints "hello, World!"
print lcfirst(uc("Hello, World!"));     # prints "hELLO, WORLD!"
```

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

5.9.1.3 chop() and chomp()

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

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

use strict;
```

```
my $char = chop("Hello");               # $char is now equal to "o"

my $string = "Goodbye";

$char = chop $string;
print $char . "\n";                     # "e"
print $string . "\n";                   # "Goodby"
```

The `chomp()` works similarly, but *only* removes the last character if it is a newline. This is very handy for removing extraneous newlines from user input.

5.9.1.4 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.

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

use strict;

my $string = "Hello, world!";
print substr($string, 0, 5);             # prints "Hello"

substr($string, 0, 5) = "Greetings";
print $string;                           # prints "Greetings, world!"
```

5.9.2 Numeric functions

There are many numeric functions in Perl, including trig 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)

5.9.3 Type conversions

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

- `oct()`

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

5.9.4 Manipulating lists and arrays

5.9.4.1 Stacks and queues

Stacks and queues are special kinds of lists.

A stack can be thought of like a stack of paper on a desk. Things are put onto the top of it, and taken off the top of it.

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

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

- `push()` -- add items to the end of a list
- `pop()` -- remove items from the end of a list
- `shift()` -- remove items from the start of a list
- `unshift()` -- add items to the start of a list

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

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

5.9.4.2 Sorting lists

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

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

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

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

5.9.4.3 Converting lists to strings, and vice versa

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

5.9.5 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.

5.9.6 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.

5.9.7 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.

5.9.8 Exercises

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

1. Create a scalar variable containing the phrase "There's more than one way to do it" then print it out in all upper-case (Answer: `exercises/answers/tmtowtdi.pl`)
2. Print a random number
3. Print a random item from an array (Answer: `exercises/answers/quotes.pl`)
4. 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`)
5. Print out the date for a week ago (the answer's in `exercises/answers/lastweek.pl`
6. Print out a sentence in reverse

a. reverse the whole sentence

b. reverse just the words

(Answer: `exercises/answers/reverse.pl`)

5.10 Chapter summary

- Perl operators and functions can be used to manipulate data and perform other necessary tasks
- The difference between operators and functions is blurred; most can behave in either way
- Chapter 3 of your Camel book, **perldoc perlop**, **perldoc perlfunc**, and **perldoc -f *`functionname`*** can be used to find out detailed information about operators and functions.
- Functions can accept arguments of various kinds
- Functions may return scalars, lists etc
- Return values may differ depending on whether a function is called in scalar or list context

Chapter 6: Conditional constructs

In this chapter...

In this section, 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.

6.1 What is a block?

The simplest block is a single statement, for instance:

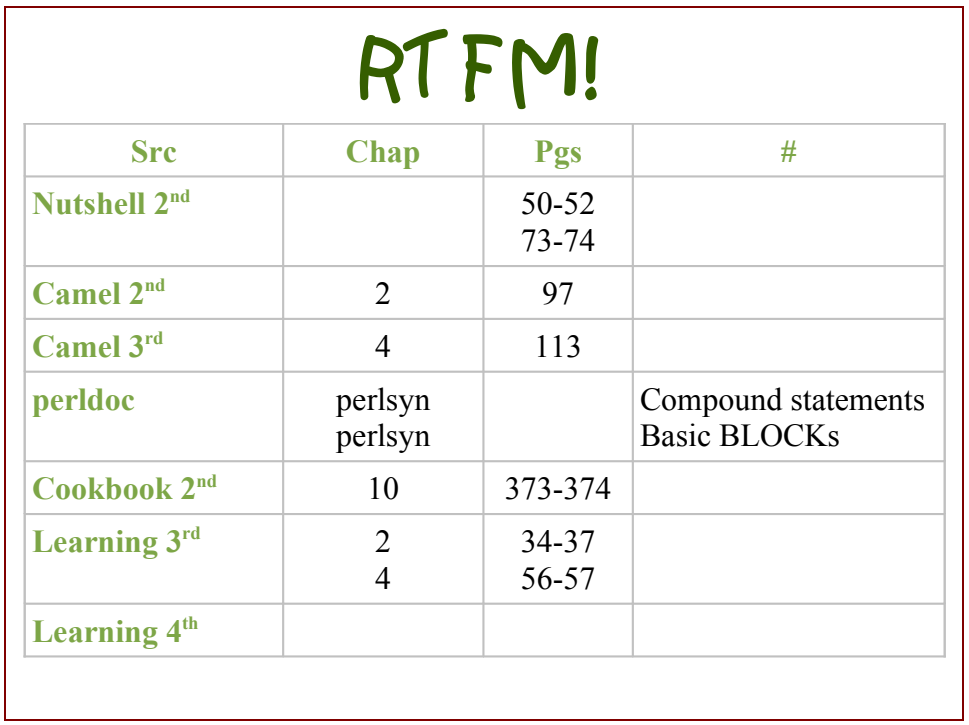
```
print "Hello, world!\n";
```

Sometimes you'll want several statements to be grouped together logically. That's what we call a block. A block can be executed either in response to some condition being met, or as an independent chunk of code that's 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.

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

You'll notice that the body of the block is indented from the brackets; this is to improve readability. Make a habit of doing it.



Src	Chap	Pgs	#
Nutshell 2nd		50-52 73-74	
Camel 2nd	2	97	
Camel 3rd	4	113	
perldoc	perlsyn perlsyn		Compound statements Basic BLOCKs
Cookbook 2nd	10	373-374	
Learning 3rd	2 4	34-37 56-57	
Learning 4th			

6.2 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*

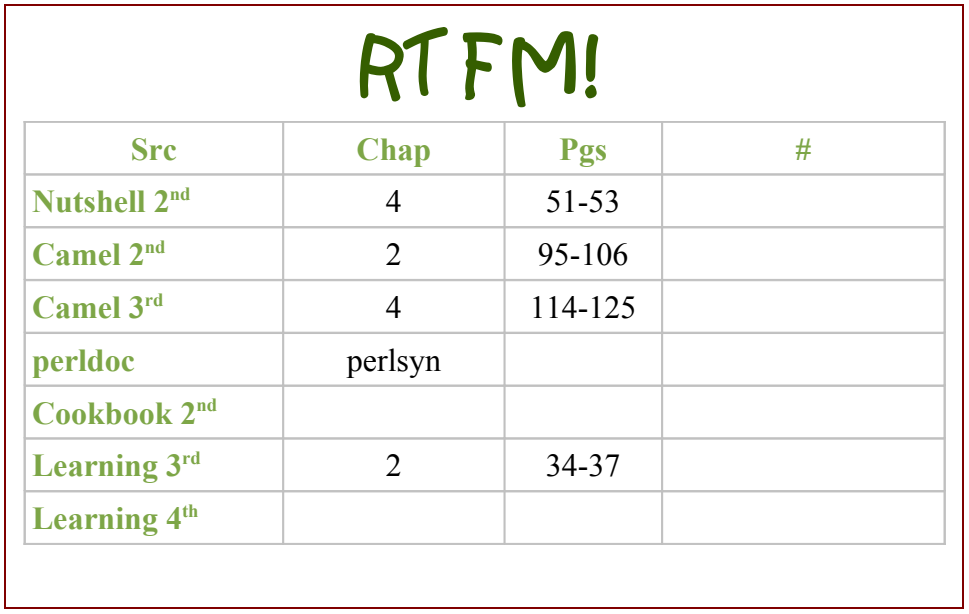
```
#!/usr/bin/perl -w

use strict;

my $a = "foo";

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

print $a;                       # prints foo
```

Now, onto the situations in which we'll encounter blocks.

6.3 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, so we'll just assume that you get the general idea.

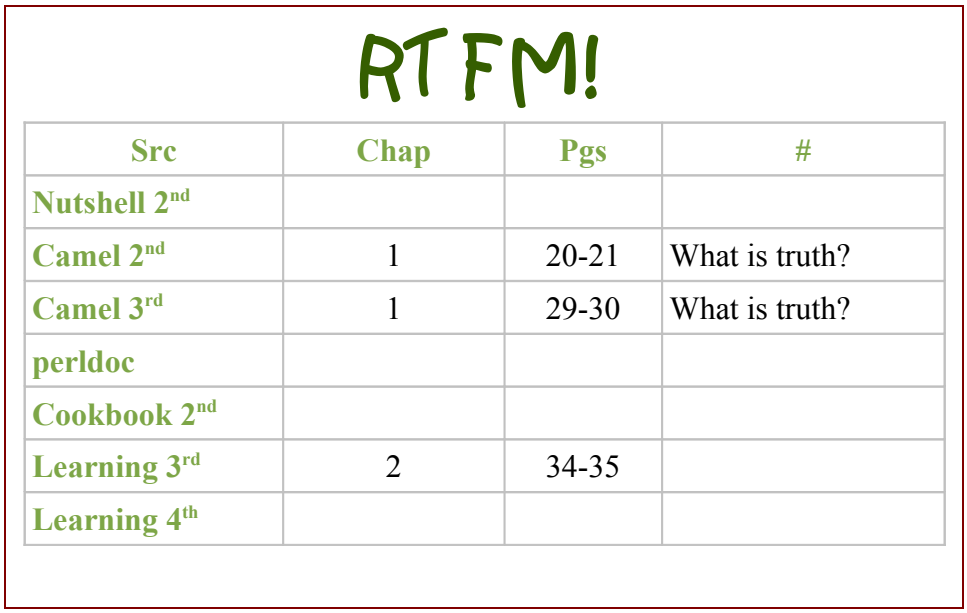


Src	Chap	Pgs	#
Nutshell 2nd	4	51-53	
Camel 2nd	2	95-106	
Camel 3rd	4	114-125	
perldoc	perlsyn		
Cookbook 2nd			
Learning 3rd	2	34-37	
Learning 4th			

6.4 What is truth?

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

```
42              # true
0               # false
"0"             # false, because perl switches it to a number when
it
                # needs to
"wibble"        # true
$new_variable   # false (if we haven't set it to anything, it's
                # undefined)
```



Src	Chap	Pgs	#
Nutshell 2nd			
Camel 2nd	1	20-21	What is truth?
Camel 3rd	1	29-30	What is truth?
perldoc			
Cookbook 2nd			
Learning 3rd	2	34-35	
Learning 4th			

6.5 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 6-1. 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

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

Table 6-2. String comparison operators

Operator	Meaning
`eq`	Equality
`ne`	Inequality
`lt`	Less than (in "asciibetical" order)
`gt`	Greater than
`le`	Less than or equal to
`ge`	Greater than or equal to

Some examples:

```
69 > 42                     # true
"0" == 3 - 3                # true
'apple' gt 'banana'         # false; apple is alphabetically before
                            # banana
1 + 2 == "3com"             # true - 3com is evaluated in numeric
                            # context because we used == not eq
```

Assigning `undef` to a variable name undefines it again, as does using the `undef`

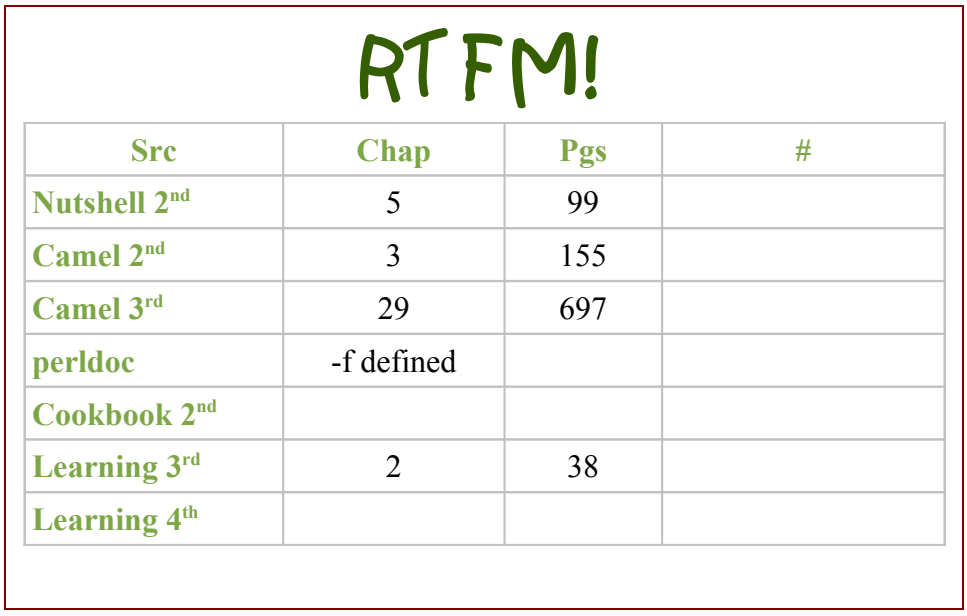
function with the variable's name as its argument.

6.5.1 Existence and Defined-ness

We can also check whether things are defined (something is defined when it's 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. You can't just use the name of the variable because the variable can be defined an still evaluate to false - for example, if you assign it the value `0`.

```
$skippy = "bush kangaroo";
if (defined($skippy)) {
        print "Skippy is defined.\n";
} else {
        print "Skippy is undefined.\n";
}
```

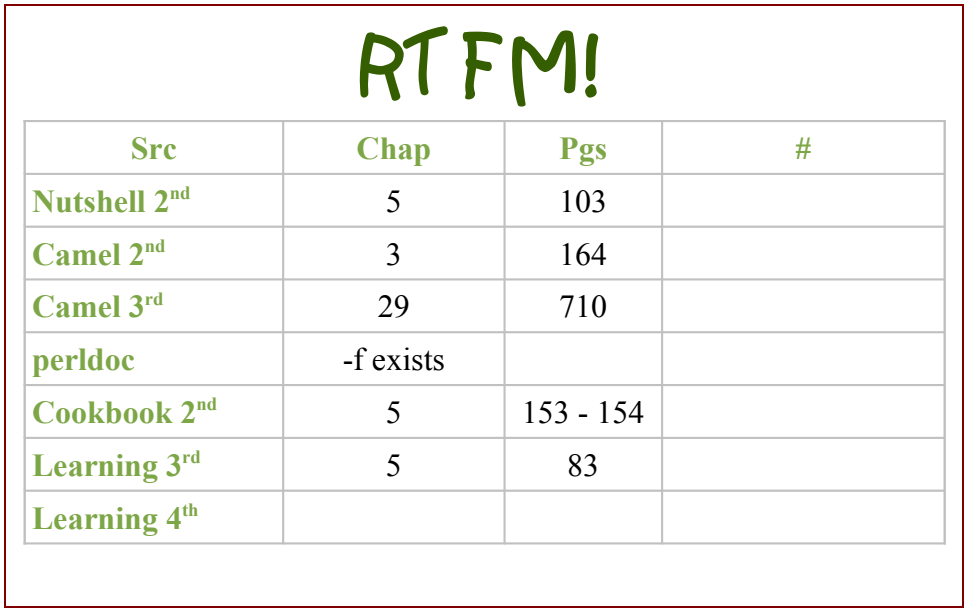


Src	Chap	Pgs	#
Nutshell 2nd	5	99	
Camel 2nd	3	155	
Camel 3rd	29	697	
perldoc	-f defined		
Cookbook 2nd			
Learning 3rd	2	38	
Learning 4th			

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

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

if (exists($animals{"Blinky Bill"}) {
```

```
        print "Blinky Bill exists.\n";
} else {
        print "Blinky Bill doesn't exist.\n";
}
```



Src	Chap	Pgs	#
Nutshell 2nd	5	103	
Camel 2nd	3	164	
Camel 3rd	29	710	
perldoc	-f exists		
Cookbook 2nd	5	153 - 154	
Learning 3rd	5	83	
Learning 4th			

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

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

if ( exists( $miscellany{"wombat"} ) ) {      # doesn't exist
        print "Wombat exists\n";
} else {
        print "We have no wombats here.\n";   # this will happen
}
```

6.5.2 Boolean logic operators

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

The short circuit operators come in two flavours: line noise, and English. Both do similar things but have different precedence. This causes great confusion.

There are two ways of avoiding this: use lots of brackets, or read page 89 of the Camel book very, very carefully.

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

Table 6-3. Boolean logic operators

English-like	C-style	Example	Result
`and`	`&&`	`$a && $b`	True if both `$a` and `$b` are true; acts on `$a` then if `$a` is true, goes on to act on `$b`.
`or`	`||`	`$a || $b`	True if either of `$a` and `$b` are true; acts on `$a` then if `$a` is false, goes on to act on `$b`.

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

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

$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 (parenthesized expression
                                    # evaluated first)
```

6.5.3 Using boolean logic operators as short circuit operators

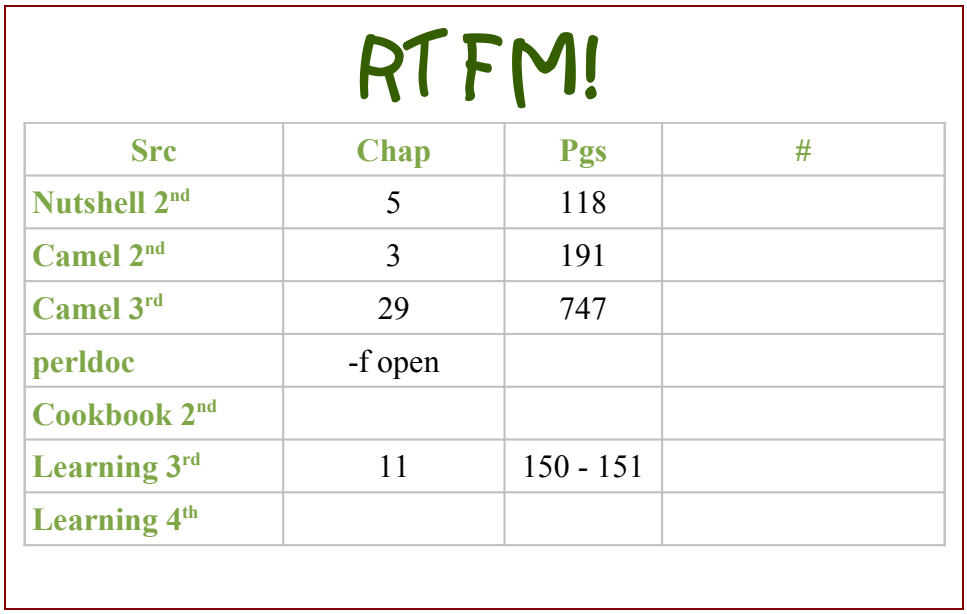
These operators aren't just for combining tests in conditional statements --- they

can be used to combine other statements as well.

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

```
open(INFILE, "input.txt") or die("Can't open input file: $!");
```

What is it doing?



Src	Chap	Pgs	#
Nutshell 2nd	5	118	
Camel 2nd	3	191	
Camel 3rd	29	747	
perldoc	-f open		
Cookbook 2nd			
Learning 3rd	11	150 - 151	
Learning 4th			

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') and print "Have a good weekend!\n";
```

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

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

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

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

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

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd			
Camel 2nd	2	89	short circuit operators
Camel 3rd	3	102	
perldoc			
Cookbook 2nd			
Learning 3rd	10	143	
Learning 4th			

6.6 Types of conditional constructs

You'll have noticed that we snuck in something new in the last section -- the `if` construct. It probably didn't surprise you much - you'll have seen something similar in just about every programming language. (Bonus points will *not* be given for naming programming languages which have no "if" construct.)

6.6.1 if statements

The `if` construct goes like this:

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

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

If you're testing for something negative, it can sometimes make sense to use the similar-but-opposite construct, `unless`.

```
unless (conditional statement) {
```

```
        # BLOCK
}
```

There is no such thing as an "elsunless" (thank the gods!), 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 "Yes, we have no bananas\n" unless $bananas;
```

6.6.2 while loops

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

```
while (conditional statement) {
        # BLOCK
}

my $hunger = 5;
while ($hunger) {
        print "Feed me!\n";
        $hunger--;
}
```

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

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

6.6.3 for and foreach

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

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

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

```
# using a for loop

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

# using foreach

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

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

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

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

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

We'll look at hash functions later.

6.6.4 Exercises

1. Set a variable to a numeric value, then create 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"
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.
3. Use a `while` loop to print out a numbered list of the elements in an array

4. Now do it with a `foreach` loop
5. Now do it with a hash, printing out the keys and values for each item (hint: look up the `keys` function in your Camel book)

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

6.7 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 any file it has redirected to its STDIN:

```
$ cat < hello.pl
```

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

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

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

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

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

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

Conveniently enough, the `while` statement can be written more succinctly, because in these circumstances, the line input operator assigns to `$_` by default:

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

Better yet, the default filehandle used by the line input operator is STDIN, so we can shorten the above example yet further:

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

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

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.

Try running the script with no arguments. You'll have to type some stuff in, line by line, and type **CTRL-D** (a.k.a. `^D`) when you're ready to stop. `^D` indicates end-of-file (EOF) on most Unix systems.

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.

6.8 Best practices template for file manipulation

Its a good idea to follow this template when reading and writing from files:

```
my $filename = 'filename'; # the filename

my $fh;

open($fh, "<", $filename) or die "couldn't open $filename for read
($!)";

while(my $line = <$fh>) {
      chomp($line);
      # do whatever else you want to do with it
}

close($fh) or die "couldn't close $filename ($!)";
```

There are a couple of points to note about this. The first would be the use of the 3-argument open(). Another would be stooring the filename in a scalar for use in error messages. die()ing on open() and close() is considered good form and the system-provided error ($!) can be very helpful.

6.9 Named blocks

Blocks can be given names, thus:

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

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

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

6.10 Breaking out of loops

You can break out of loops using `next`, `last` and similar statements.

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

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

The `LINE` indicating the block to break out of is optional (it defaults to the current smallest loop), but can be very useful when you wish to break out of a loop higher up the chain:

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

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

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

6.11 Chapter summary

- A block in Perl is a series of statements grouped together by curly brackets. 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 null string, or undefined
- The `if - elsif - else` conditional construct can be used to perform certain actions based on the truth of a condition
- 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 break out of blocks using `next`, `last` and similar statements

Chapter 7: Subroutines

In this chapter...

In this chapter, we look at subroutines and how they can be used to simplify your code.

7.1 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. 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, linecount).

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
}

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

7.2 Calling a subroutine

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

```
&print_headers;
print_headers();
```

If (for some reason) you've got a subroutine that clashes with a reserved function or something, you will need to prefix your function name with `&` (ampersand) to be perfectly clear. You should avoid doing this anyway; overloading built-in functions can cause more confusion than it's worth.

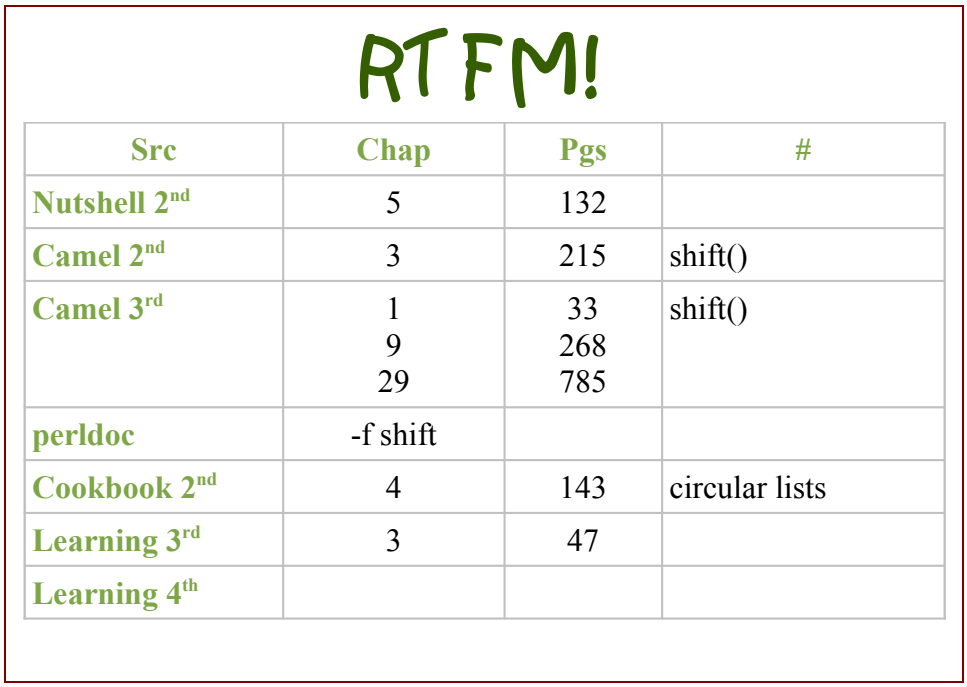
> **ADVANCED**
>
> There are other times when you need to use an ampersand on your subroutine name, such as when a function needs a SUBROUTINE type of parameter, or when making an anonymous subroutine reference.

7.3 Passing arguments to a subroutine

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

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

# we can also pass variables to a subroutine by name...
my $fiction_title = "Lord of the Rings";
my $fiction_author = "J.R.R. Tolkein";
print_headers($fiction_title, $fiction_author);

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

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



Src	Chap	Pgs	#
Nutshell 2nd	5	132	
Camel 2nd	3	215	shift()
Camel 3rd	1 9 29	33 268 785	shift()
perldoc	-f shift		
Cookbook 2nd	4	143	circular lists
Learning 3rd	3	47	
Learning 4th			

7.4 Returning values from a subroutine

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

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

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

You can also return lists from your subroutine:

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

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

7.5 Exercises

1. Write a subroutine which prints out its first argument
2. Modify the above subroutine to also print out the last argument
3. Now change it to compare the first and last arguments and return the one which is numerically larger (you'll want to use an `if` statement for that)

7.6 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 `@_`
- Subroutines can return scalar or list values.

Chapter 8: Regular expressions

In this chapter...

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

8.1 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 wildcards - `*.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 any kind of text can be matched, and that the set of special characters is different.

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

- If you have a mathematical background, you may like to think of a regexp as a definition of a set of strings. For instance, a regexp may describe the set of all strings which begin with the letter "a".

8.2 Regular expression operators and functions

8.2.1 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 parenthesized.
- 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!http://!;  # using ! as an alternative delimiter
}
```

8.2.2 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.
- You can use any delimiter you want, the same as the `m//` operator.

- Using `/g` on the end of it matches globally, otherwise matches (and replaces) only the first instance of the pattern.
- Using the `/i` modifier makes it case insensitive.

```
# fix some misspelt text

while (<>) {
        s/freind/friend/g;
        s/teh/the/g;
        s/jsut/just/g;
        print;
}
```

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

8.2.3 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 8-1. 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 =~ /geocities/) {
        print "Ahhh, I see you have a geocities homepage!\n";
}
```

8.3 Metacharacters

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

8.3.1 Some easy metacharacters

Table 8-2. Regular expression metacharacters

Metacharacter(s)	Matches...
`^`	Start of string
`$`	End of string
`.`	Any single character except `\n` (though special things can happen in multiline mode)
`\n`	Newline (subtly different to `$` - when working in multiline mode, there may be newlines embedded in the multiline string you're working with.
`\t`	Matches a tab
`\s`	Any whitespace character, such as space or tab
`\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.

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd	4	67 - 73	
Camel 2nd	2	58 - 68	
Camel 3rd	5	158 - 164	
perldoc	perlre		
Cookbook 2nd			
Learning 3rd	7	100	
Learning 4th			

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

Some quick examples:

```
# Perl regular expressions are usually found within slashes - the
# matching operator/function which we will see soon.

/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

# 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
```

8.3.2 Quantifiers

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

Table 8-3. 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

8.3.3 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".

To change this behaviour, follow the quantifier with a question mark, for example `.*?`. This is sometimes referred to as "minimal matching".

```
$string = "abracadabra";

/a.*a/                # greedy -- matches "abracadabra"
/a.*?a/               # not greedy -- matches "abra"
```

8.3.4 Exercises

1. You now know enough to work out the price example above. Work it through.
2. Another example: what regular expression would match the word "colour" with either British or American spellings?
3. How can we match any four-letter word?

8.4 Grouping techniques

8.4.1 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 occurences 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.

We can negate a character class by putting a caret at the start of it. That's right, the same character that we used to match the start of the line. Larry Wall has written that Perl does anything you want -- unless you want consistency, and it has also been said that consistency is the hobgoblin of small minds. Therefore, we'll learn about these character class inconsistencies, learn to love them, and flatter ourselves that we do not have small minds.

Here are some of the special rules that apply inside character classes. I make no guarantee that this is a complete list; additions are always welcome.

- `^` at the start of a character class negates the character class, rather than specifying the start of a line.
- `-` specifies a range of characters.
- `$ . () \{\ * +}` and other metacharacters taken literally.

8.4.1.1 Exercises

Your trainer 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?

8.4.2 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:

```
/cat|dog|budgie/                # matches any of our pets
```

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

```
/^cat|dog|budgie$/
```

...to match any of our pets on a line by itself, what we're actually matching is: "the start of the string followed by cat; or dog; or budgie followed by the end of the string". This is not what we originally intended. To fix this, we enclose our alternation in round brackets:

```
/^(cat|dog|budgie)$/

# 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`.

8.4.3 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!). What we use it to mean is "something that is a chunk of regular expression in its own right" -- as opposed to "something that can wipe out cities with a single blast".

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 (but we'll deal with that later).

In the example above, there are three atoms:

1. start of line
2. cat or dog or budgie
3. end of line

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

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

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

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

8.5 Exercises

1. Determine whether your name appears in a string (an answer's in `exercises/answers/namere.pl`).
2. Remove footnote references (like [1]) from some text (see `exercises/footnote.txt` for some sample text, and `exercises/answers/footnote.pl` for an answer).
3. Split tab-separated data into an array then print out each element using a `foreach` loop.

8.6 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 `=~` binding operator
- Functions such as `split()` and `grep()` use regular expression patterns as one of their arguments

Chapter 9: Practical exercises

This chapter provides you with some broader exercises to test your new Perl skills. Each exercise requires you to use a mixture of variables, operators, functions, conditional and looping constructs, and regular expressions.

9.1 Exercises

There are no right or wrong answers. Remember, "There's More Than One Way To Do It."

1. Write a simple menu system where the user is repeatedly asked to choose a message to display or Q to quit.
   a. Consider case-sensitivity
   b. Handle errors cleanly
2. Write a "chatterbox" program that holds a conversation with the user by matchings patterns in the user's input.
3. Write a program that gives information about files.
   a. use file test operators
   b. offer to print the file out if it's a text file
   c. how will you cope with files longer than a screenful?

Chapter 10: File I/O

In this chapter...

In this section, we learn how to open and interact with files and directories in various ways.

10.1 Assumed knowledge

You should already have encountered the `open()` function and the `<>` line input operator in a previous Perl training session or in your previous Perl experience.

10.2 Angle brackets - the line input and globbing operators

You will have encountered the line input operator <> before, in situations such as these:

```
# reading lines from STDIN
while (<>) {
        ...
        ...
}

# reading a single line of user input from STDIN
my $input = <STDIN>
```

The line input operator is discussed in-depth on page 53 of the Camel. Read it now.

Src	Chap	Pgs	#
Nutshell 2nd			
Camel 2nd			
Camel 3rd			
perldoc			
Cookbook 2nd			
Learning 3rd			
Learning 4th			

- 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.
- The default filehandle is `STDIN`, or any files listed on the command line of the Perl script (eg **myscript.pl file1 file2 file3**).

The *globbing* operator is nearly, but not quite, identical to the line input operator. It looks the same, and it acts partly in a similar way, but it really is a separate operator.

The filename globbing operator is documented on page 55 of the Camel.

If the angle brackets have anything in them other than a filehandle or nothing, it will work as a globbing operator and whatever is between the angle brackets will be treated as a filename wildcard. For instance:

```
my @files = <*.txt>
```

The filename glob `*.txt` is matched against files in the current directory, then either they are returned as a list (in list context, as above) or one scalar at a time (in scalar context).

If you get a list of files this way, you can then open them in turn and read from them.

```
while (<*.txt>) {
        open (FILEHANDLE, $_) || die ("Can't open $_: $!");
        ...
        ...
        close FILEHANDLE;
}
```

The `glob()` function behaves in a very similar manner to the angle bracket globbing operator.

```
my @files = glob("*.txt")
```

```
foreach (glob "*.txt") {
        ...
}
```

The `glob()` is considered much cleaner and better to use than the angle-brackets globbing operator.

10.2.1 Exercises

1. Use the line input operator to accept input from the user then print it out
2. Modify your previous script to use a `while` loop to get user input repeatedly, until they type "Q" (or "q" - check out the `lc()` and `uc()` functions in chapter 3 of your Camel book) (Answer: `exercises/answers/userinput.pl`)
3. Use the file globbing function or operator to find all Perl scripts in your home directory and print out their names (assuming they are named in the form `*.pl`) (Answer: `exercises/answers/findscripts.pl`)

10.2.1.1 Advanced exercises

1. Use the above example of globbing to print out all the Perl scripts one after the other. You will need to use the `open()` function to read from each file in turn. (Answer: `exercises/answers/printscripts.pl`)

10.3 open() and friends - the gory details

10.3.1 Opening a file for reading, writing or appending

The `open()` function is used to open a file for reading or writing (or both, or as a pipe - more on that later).

The `open()` function is documented on pages 191-195 of the Camel book, and also in **perldoc perlfunc**. Read the documentation for `open()` before going any further.

Src	Chap	Pgs	#
Nutshell 2nd			
Camel 2nd			
Camel 3rd			
perldoc			
Cookbook 2nd			
Learning 3rd			
Learning 4th			

In a typical situation, we might use `open()` to open and read from a file:

```
open(LOGFILE, "/var/log/httpd/access.log")
```

Note that the `<` (less than) used to indicate reading is assumed; we could equally well have said `"</var/log/httpd/access.log"`.

You should *always* check for failure of an `open()` statement:

```
open(LOGFILE, "/var/log/httpd/access.log") || die "Can't open
        /var/log/httpd/access.log: $!";
```

`$!` is the special variable which contains the error message produced by the last system interaction. It is documented in chapter 2 of the Camel, on page 134.

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd			
Camel 2nd			
Camel 3rd			
perldoc			
Cookbook 2nd			
Learning 3rd			
Learning 4th			

Once a file is opened for reading or writing, we can use the filehandle we specified (in this case `LOGFILE`) for a variety of useful purposes:

```
open(LOGFILE, "/var/log/httpd/access.log") || die "Can't open
        /var/log/httpd/access/log: $!";

# use the filehandle in the in the <> line input operator...
while (<LOGFILE>) {
        print if /PerlClass.com.com.au/;
}

close LOGFILE;

# open a new logfile for appending
open(SCRIPTLOG, ">>myscript.log") || die "Can't open myscript.log: $!";

# print() takes an optional filehandle argument - defaults to STDOUT
print SCRIPTLOG "Opened logfile successfully.\n";

close SCRIPTLOG;
```

Note that you should always close a filehandle when you're finished with it (though admittedly any open filehandles will be automatically closed when your script exits).

You can also use `sysopen()` and friends to open a file in a C-like way. See page 229 of your Camel book for details or **perldoc -f sysopen**.

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd			
Camel 2nd			
Camel 3rd			
perldoc			
Cookbook 2nd			
Learning 3rd			
Learning 4th			

10.3.1.1 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. (Answer: `exercises/answers/100count.pl`)
4. Create a new script which opens a logfile for appending. Create a `while` loop which accepts input from STDIN and appends each line of input to the logfile. (Answer: `exercises/answers/logfile.pl`)
5. Create a script which opens two files, reads input from the first, and writes it out to the second. (Answer: `exercises/answers/readwrite.pl`)

10.3.2 Reading directories

It is also possible to open directories (using `opendir()` and read from them. However, it is not possible to read the contents of files in that directory simply by opening it and looping through it. Opening a directory simply makes the file-names in that directory accessible via functions such as `readdir()`.

`opendir()` is documented on page 195 of the Camel. `readdir()` is on page 202. Don't forget that function help is also available by typing **perldoc -f opendir** or **perldoc -f readdir**


```
opendir(HOMEDIR, $ENV{HOME});

my @files = readdir(HOMEDIR);

closedir HOMEDIR;

foreach (@files) {
        open(THISFILE, "<$_") || die "Can't open file $_: $!");
        ...
        ...
        close THISFILE;
}
```

10.3.2.1 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`)

10.3.3 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, however, as `+>` would overwrite the file before you had a chance to read from it.

Read/write access to a file is not as useful as it sounds --- you can't 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.

A more flexible way to read and write a file is to import the file into an array, manipulate the array, then output each element again.

```
# program to remove duplicate lines
open(INFILE, "file.txt") || die "Can't open file.txt for input: $!";
my @lines = <INFILE>;
close INFILE;

# dup-remover taken from The Perl Cookbook
my @unique = grep { ! $seen{$_} ++ } @lines;

open(OUTFILE, ">file.txt") || die "Can't open file.txt for output: $!";
foreach (@unique) {
        print OUTFILE $_;
}

close OUTFILE;
```

- One thing to watch out for here is memory usage. If you have a ten megabyte file, it will use at least that much memory as a Perl data structure.

10.3.3.1 Exercises

1. Open a file, reverse its contents (line by line) and write it back to the same filename (Answer: `exercises/answers/reversefile.pl`)

10.3.4 Opening pipes

If the filename given to `open()` begins with a pipe symbol (`|`), the filename is interpreted as a command to which output is to be piped, and if the filename ends with a `|`, the filename is to be interpreted as a filename which pipes input to us.

This is often used when you want to take input from the system a line at a time. Here's an example which reads from the **rot13** filter (a simple routine which rotates the letters of its input by 13 letters, providing a very simple cipher for encoding the answers to jokes, spoilers to movies, or other low-security information):

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

use strict;

open (ROT13, "rot13 < /etc/motd |") || die "Can't open pipe: $!";

while (<ROT13>) {
        print;
```

```
}

close ROT13;
```

Conversely, we can output something through rot13:

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

use strict;

open (ROT13, "|rot13") || die "Can't open pipe: $!";

print "This is some rot13'd text:\n";
print ROT13 "This is some rot13'd text.\n";

close ROT13;
```

If you reverse the two print lines above, the output will nevertheless be in the same order as before. You'll need to set `$|` to flush the output pipe. It's on page 130 of your Camel, or in **perldoc perlvar**.

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd			
Camel 2nd			
Camel 3rd			
perldoc			
Cookbook 2nd			
Learning 3rd			
Learning 4th			

10.3.4.1 Exercises

1. Modify the second example above (provided for you as `exercises/rot13.pl` in your exercises directory to accept user input and print out the **rot13**'d version.
2. Change your script to accept input from a file using `open()` (Answer: `exercises/answers/rot13.pl`)

3. Change your script to pipe its input through the **strings** command, so that if you get a file that's not a text file, it will only look at the parts of the file which are strings. (Answer: `exercises/answers/strings.pl`)

10.4 Finding information about files

We can find out various information about files by using file test operators and functions such as `stat()`

Table 2-1. 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.
`-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.
`-B`	File is a binary file (opposite of -T).
`-M`	Age of file in days when script started.
`-A`	Same for access time.
`-C`	Same for inode change time.

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd	4	63 - 64	
Camel 2nd	2	85	
Camel 3rd	3	98	
perldoc	perlfunc		
Cookbook 2nd			
Learning 3rd	11	157 - 163	
Learning 4th			

Here's how the file test operators are usually used:

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

use strict;

unless (-e "config.txt") {
        die "Config file doesn't exist";
}

# or equivalently...
die "Config file doesn't exist" unless -e config.txt;
```

The `stat()` function returns similar information for a single file, in list form. `lstat()` can also be used for finding information about a file which is pointed to by a symbolic link.

10.4.1 Exercises

1. 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`)
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`)

10.5 Recursing down directories

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.

The **File::Find** module is documented in chapter 7 of the Camel, on page 439, or in **perldoc File::Find**.

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd	8	254	
Camel 2nd	7	439	
Camel 3rd	31	867	
perldoc	File::Find		
Cookbook 2nd	9	359 - 361	
Learning 3rd	12	173	pretty light
Learning 4th			

File::Find emulates Unix's **find** command. It takes as its arguments a block to execute for each file found, and a list of directories to search.

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

use strict;
use File::Find;

print "Enter the directory to search: ";
chomp(my $dir = <STDIN>);

find (\&wanted, $dir);

sub wanted {
        print "$_\n";
}
```

For each file found, certain variables are set. `$File::Find::dir` is set to the current directory name, `$File::Find::name` contains the full name of the file, i.e. `$File::Find::dir/$_`.

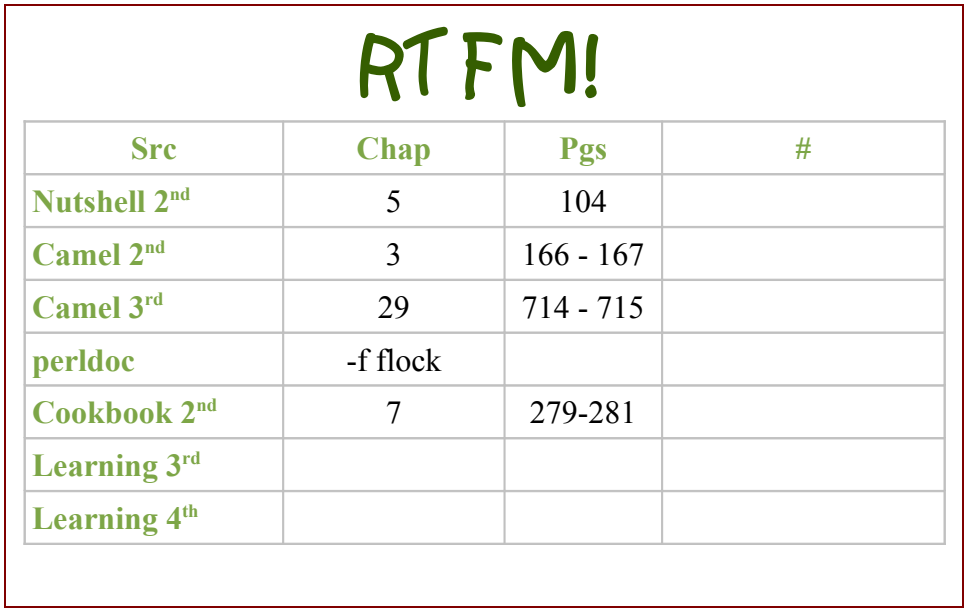
10.5.1 Exercises

1. Modify the simple script above (in your scripts directory as `exercises/find.pl`) to only print out the names of plain text files only (hint: use file test operators)
2. Now use it to print out the contents of each text file. You'll probably want to pipe your output through **more** so that you can see it all. (Answer: `exercises/answers/find.pl`)

10.6 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) users open the same file in read/write mode.

`flock()` is documented on page 166 of the Camel book, or use **perldoc -f flock** to read the online documentation.

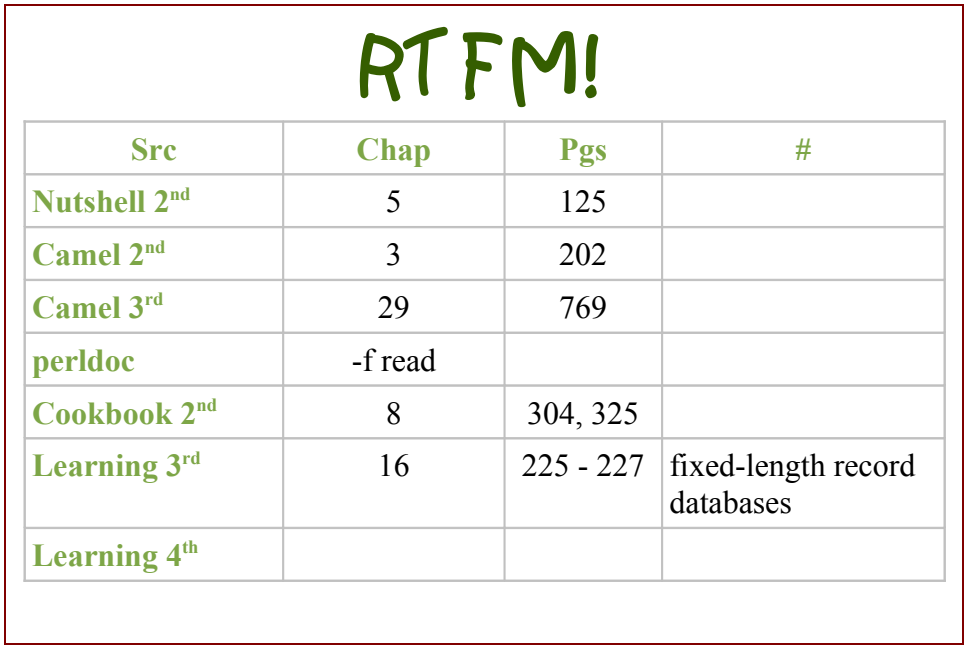


Src	Chap	Pgs	#
Nutshell 2nd	5	104	
Camel 2nd	3	166 - 167	
Camel 3rd	29	714 - 715	
perldoc	-f flock		
Cookbook 2nd	7	279-281	
Learning 3rd			
Learning 4th			

10.7 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.

Instead, we use `read()` to read a certain number of bytes from a file handle.



Src	Chap	Pgs	#
Nutshell 2nd	5	125	
Camel 2nd	3	202	
Camel 3rd	29	769	
perldoc	-f read		
Cookbook 2nd	8	304, 325	
Learning 3rd	16	225 - 227	fixed-length record databases
Learning 4th			

`read()` takes the following arguments:

- The filehandle to read from
- The scalar to put the binary data into
- The number of bytes to read
- The byte offset to start from (defaults to 0)

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

use strict;

my $image = "picture.gif";

open (IMAGE, $image) or die "Can't open image file: $!";
open (OUT, ">backup/$image") or die "Can't open backup file: $!";
```

```
my $buffer;

binmode IMAGE;

while (read IMAGE, $buffer, 1024) {
        print OUT $buffer;
}

close IMAGE;
close OUT;
```

- If you are using Windows, DOS, or some other types of systems, you may need to use `binmode()` to make sure that certain linefeed characters aren't translated when Perl reads a file in binary mode. While this is not needed on Unix systems, it's a good idea to use it anyway to enhance portability.

10.8 Chapter summary

- Angle brackets `<>` can be used for simple line input. In scalar context, they return the next line; in list context, all remaining lines; the default filehandle is `STDIN` or any files mentioned in the command line (ie `@ARGV`).
- Angle brackets can also be used as a globbing operator if anything other than a filehandle name appears between the angle brackets. In scalar context, returns the next file matching the glob pattern; in list context, returns all remaining matching files.
- The `open()` and `close()` functions can be used to open and close files. Files can be opened for reading, writing, appending, read/write, or as pipes.
- The `opendir()`, `readdir()` and `closedir()` functions can be used to open, read from, and close directories.
- The **File::Find** module can be used to recurse down through directories.
- File test operators or `stat()` can be used to find information about files
- File locking can be achieved using `flock()`
- Binary data can be read using the `read()` function. The `binmode()` function should be used to ensure platform independence when reading binary data.

Chapter 11: Advanced regular expressions

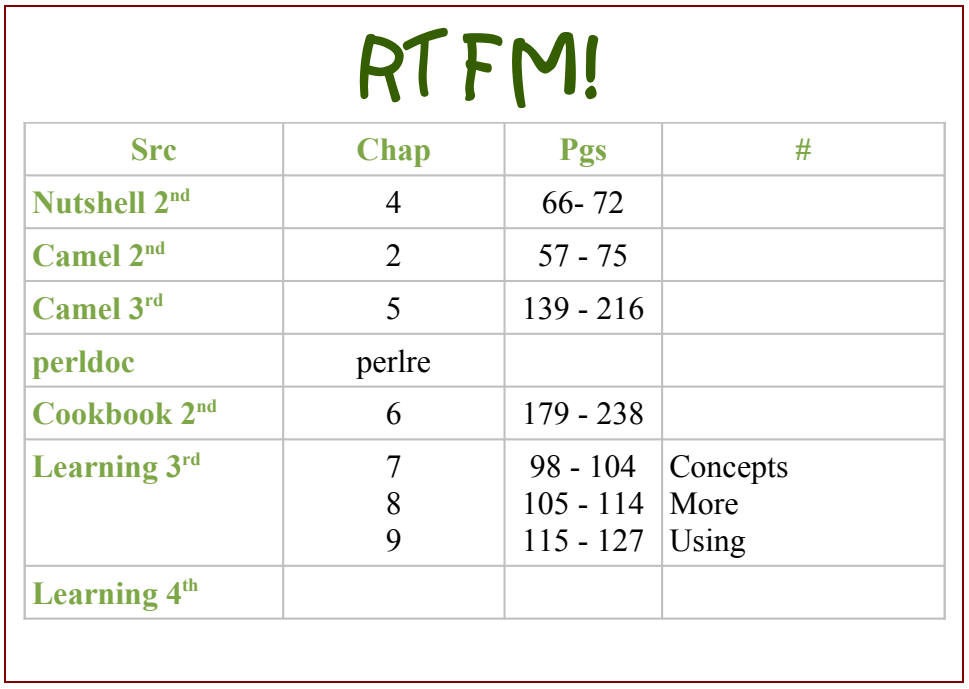
In this section...

This section builds on the basic regular expressions taught in day 1 of Perl-Class.com's *Introduction to Perl* 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.

11.1 Assumed knowledge

You should already be familiar with the following topics:

- Regular expression metacharacters
- Quantifiers
- "Greediness" in regular expressions, aka maximal and minimal matching
- Character classes and alternation
- The `m//` matching function
- The `s///` substitution function
- Matching strings other than `$_` with the `=~` matching operator
- Assigning matched strings to lvalues



Src	Chap	Pgs	#
Nutshell 2nd	4	66- 72	
Camel 2nd	2	57 - 75	
Camel 3rd	5	139 - 216	
perldoc	perlre		
Cookbook 2nd	6	179 - 238	
Learning 3rd	7 8 9	98 - 104 105 - 114 115 - 127	Concepts More Using
Learning 4th			

11.2 Review exercises

The following exercises are intended to refresh your memory of basic regular expressions:

1. Write a script to search a file for any of the names "Yasser Arafat", "Boris Yeltsin" or "Monica Lewinsky". Print out any lines which contain these names. (Answer: `exercises/answers/namesre.pl`)
2. What pattern could be used to match any of: Elvis Presley, Elvis Aron Presley, Elvis A. Presley, Elvis Aaron Presley. (Answer: `exercises/answers/elvisre.pl`)
3. What pattern could be used to match a blank line? (Answer: `exercises/answers/blanklinere.pl`)
4. What pattern could be used to match an IP address such as `203.20.104.241`, where each part of the address is a number from 0 to 255? (Answer: `exercises/answers/ipre.pl`)

11.3 More metacharacters

Here are some more advanced metacharacters, which build on the ones already covered in the Introduction to Perl module:

Table 3-1. More metacharacters

Metacharacter	Meaning
`\B`	Match anything other than a word boundary
`\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) metacharacters until `\E`
`\E`	End of lowercase/uppercase

```
# search for the C++ computer language:

/C++/        # wrong! regexp engine complains about the plus signs
/C\+\+/      # this works
/C\Q++\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
```

11.4 Working with multiline strings

Often, you will want to read a file several lines at a time. Consider, for example, a typical Unix fortune cookie file, which is used to generate quotes for the **fortune** command:

```
%
Let's call it an accidental feature.
        -- Larry Wall
%
Linux: the choice of a GNU generation
%
When you say "I wrote a program that crashed Windows", people just
stare at you blankly and say "Hey, I got those with the system, *for
free*".
        -- Linus Torvalds
%
I don't know why, but first C programs tend to look a lot worse than
first programs in any other language (maybe except for fortran, but
then I suspect all fortran programs look like `firsts')
        -- Olaf Kirch
%
All language designers are arrogant. Goes with the territory...
        -- Larry Wall
%
We all know Linux is great... it does infinite loops in 5 seconds.
        -- Linus Torvalds
%
Some people have told me they don't think a fat penguin really
embodies the grace of Linux, which just tells me they have never
seen a angry penguin charging at them in excess of 100mph.  They'd
be a lot more careful about what they say if they had.
        -- Linus Torvalds, announcing Linux v2.0
%
```

The fortune cookies are separated by a line which contains nothing but a percent sign.

To read this file one item at a time, we would need to set the delimiter to something other than the usual `\n` - in this case, we'd need to set it to something like

`\n%\n`.

To do this in Perl, we use the special variable `$/`.

```
$/ = "\n%\n";
```

Conveniently enough, setting `$/` to `""` will cause input to occur in "paragraph mode", in which two or more consecutive newlines will be treated as the delimiter. Undefining `$/` will cause the entire file to be slurped in.

```
undef $/;
$_ = <FH>;      # whole file now here
```

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd	4	53-59	
Camel 2nd	2 7	127-140 403	
Camel 3rd	28 32	653-676 884	
perldoc	perlvar English		English provides friendlier names for special variables
Cookbook 2nd			
Learning 3rd	3	49	$_ quickly
Learning 4th			

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
```

11.4.1 Exercises

1. In your directory is a file called `exercises/linux.txt` which is a set of Linux-related fortunes, formatted as in the above example. Use multiline regular expressions to find only those quotes which were uttered by Larry Wall. (Answer: `exercises/answers/larry.pl`)

11.5 Regexp modifiers for multiline data

The `/s` and `/m` modifiers can be used to treat the string you're matching against as either a single or multiple lines. In single line mode, `^` will match only at the start of the entire string, and `$` will match only at the end of the entire string. In multiline mode, they will match at embedded newlines as well.

```
my $string = qq(
This is some text
and some more text
spanning several lines
);

if ($string =~ /^and some/m) {                  # this will match
        print "Matched in multiline mode\n";
}

if ($string =~ /^and some/s) {                  # this won't match
        print "Matched in single line mode\n";
}
```

In single line mode, the dot metacharacter will match `\n`. In multiline mode, it won't.

The differences between default, single line, and multiline mode are set out very succinctly by Jeffrey Friedl in *Mastering Regular Expressions* (see the Bibliography 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 multiline mode" refers to a mode which is similar to multi-line, but which does not strip the newline character from the end of each line.

Table 3-2. Effects of single and multiline options

Mode	Specified with	^ matches start of ...	$ matches end of ...	Dot matches newline
default	neither `/s` nor `/m`	string	string	No
single-line	`/s`	string	string	Yes
multi-line	`/m`	line	line	No
clean multi-line	`/ms`	line	line	Yes

11.6 Backreferences

11.6.1 Special variables

There are several special variables related to regular expressions.

- `$&` is the matched text
- `$`` is the unmatched text to the left of the matched text
- `$'` is the unmatched text to the right of the matched text
- `$1`, `$2`, `$3`, etc. The text matched by the 1st, 2nd, 3rd, etc sets of parentheses.

All these variables are modified when a match occurs, and can be used in any way that other scalar variables can be used.

```
# this...
my ($match) = m/^(\d+)/;
print $match;

# is equivalent to this:
m/^\d+/;
print $&;

# match the first three words...
m/^(\w+) (\w+) (\w+)/;
print "$1 $2 $3\n";
```

You can also use `$&` and other special variables in substitutions:

```
$string = "It was a dark and stormy night.";
$string =~ s/dark|wet|cold/very $&/;
```

If you want to use parentheses simply for grouping, and don't want them to set a `$1` style variable, you can use a special kind of *non-capturing* parentheses, which look like `(?: ... )`

```
# this only sets $1 - the first two sets
# of parentheses are non-capturing
m/^(?:\w+) (?:\w+) (\w+)/;
```

The special variables `$1` and so on can be used in substitutions to include matched text in the replacement expression:

```
# swap first and second words
s/^(\w+) (\w+)/$2 $1/;
```

However, this is no use in a simple match pattern, because `$1` and friends aren't set until after the match is complete. Something like:

```
my $word = "this";
print if m/($word) $1/;
```

... will *not* match "this this". Rather, it will match "this" followed by whatever `$1` was set to by an earlier match.

In order to match "this this" we need to use the special regular expression metacharacters `\1`, `\2`, etc. These metacharacters refer to parenthesized parts of a match pattern, just as `$1` does, but *within the same match* rather than referring back to the previous match.

```
my $word = "this";
print if m/($word) \1/;
```

11.6.2 Exercises

1. Write a script which swaps the first and the last words on each line (Answer: `exercises/answers/firstlast.pl`)
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`)

11.6.3 Advanced

1. Modify the above script to work across line boundaries (Answer: `exercises/answers/multiline_double.pl`)
2. What about case sensitivity?

11.7 Section summary

- Input data can be split into multiline strings using the special variable `$/`, also known as `$INPUT_RECORD_SEPARATOR`.
- The `/s` and `/m` modifiers can be used to treat multiline 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 metacharacters such as `\1`, `\2` etc may be used *within* the regular expression itself, to refer to text previously matched.

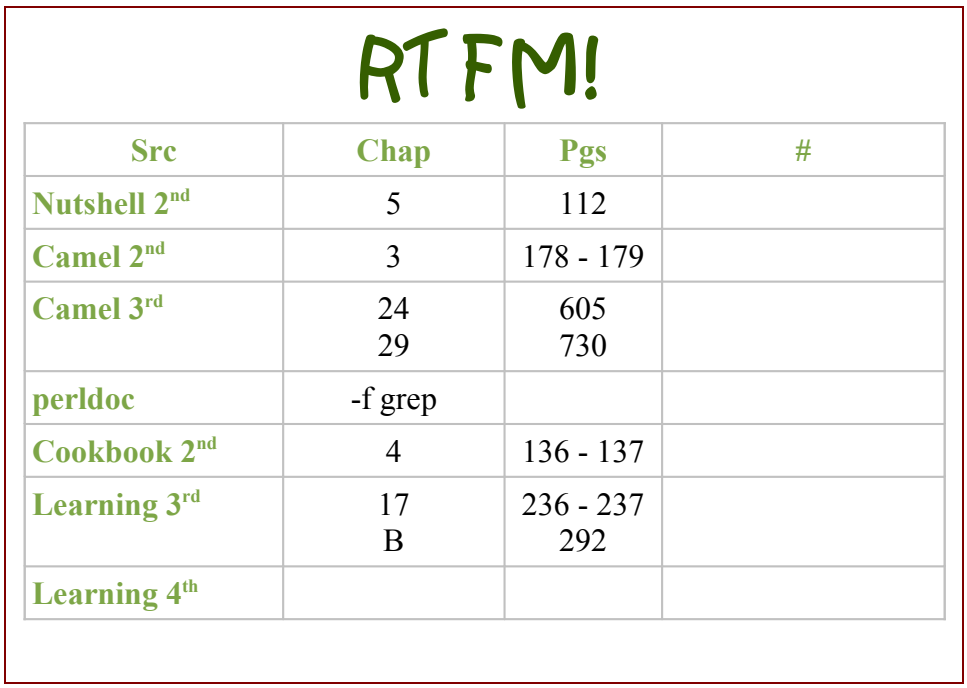
Chapter 12: More functions

In this chapter...

In this chapter, we discuss some more advanced Perl functions.

12.1 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.



Src	Chap	Pgs	#
Nutshell 2nd	5	112	
Camel 2nd	3	178 - 179	
Camel 3rd	24 29	605 730	
perldoc	-f grep		
Cookbook 2nd	4	136 - 137	
Learning 3rd	17 B	236 - 237 292	
Learning 4th			

```
# trivially check for valid email addresses
my @valid_email_addresses = grep /\@/, @email_addresses;
```

The `grep()` function temporarily assigns each element of the list to `$_` then performs matches on it.

There are many more complicated uses for the grep function. For instance, instead of a pattern you can supply an entire block which is to be used to process the elements of the list.

```
my @long_words = grep { (length($_) > 8); } @words;
```

`grep()` doesn't require a comma between its arguments if you are using a block as the first argument, but does require one if you're just using an expression. Have a look at the documentation for this function to see how this is described.

12.1.1 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`)

12.2 The map() function

The `map()` function can be used to perform an action on each member of a list and return the results as a list.

```
my @lowercase = map lc, @words;
my @doubled = map { $_ * 2 } @numbers;
```

`map()` is often a quicker way to achieve what would otherwise be done by iterating through the list with `foreach`.

```
foreach (@words) {
        push (@lowercase, lc($_);
}
```

Like `grep()`, it doesn't require a comma between its arguments if you are using a block as the first argument, but does require one if you're just using an expression.

12.2.1 Exercises

1. Create an array of numbers. Use `map()` to find the square of each number. Print out the results.

12.3 Chapter summary

- The `grep()` function can be used to find items in a list which match a certain regular expression
- The `map()` function can be used to perform an operation on each member of a list.

Chapter 13: System interaction

In this section...

In this section, 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.

13.1 system() and exec()

The `system()` and `exec()` functions both execute system commands.

`system()` forks, executes the commands given in its arguments, waits for them to return, then allows your Perl script to continue. `exec()` does not fork, and exits when it's done. `system()` is by far the more commonly used.

```
$ perl -we 'system("/bin/true");  print "Foo\n";'
Foo

$ perl -we 'exec("/bin/true");  print "Foo\n";'
Statement unlikely to be reached at -e line 1.
(Maybe you meant system() when you said exec()?)
```

If the system command fails, the error message will be available via the special variable `$!`.

```
$ perl -e 'system("cat non-existant-file") || die "$!";'
cat: non-existant-file: No such file or directory
```

13.1.1 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`)

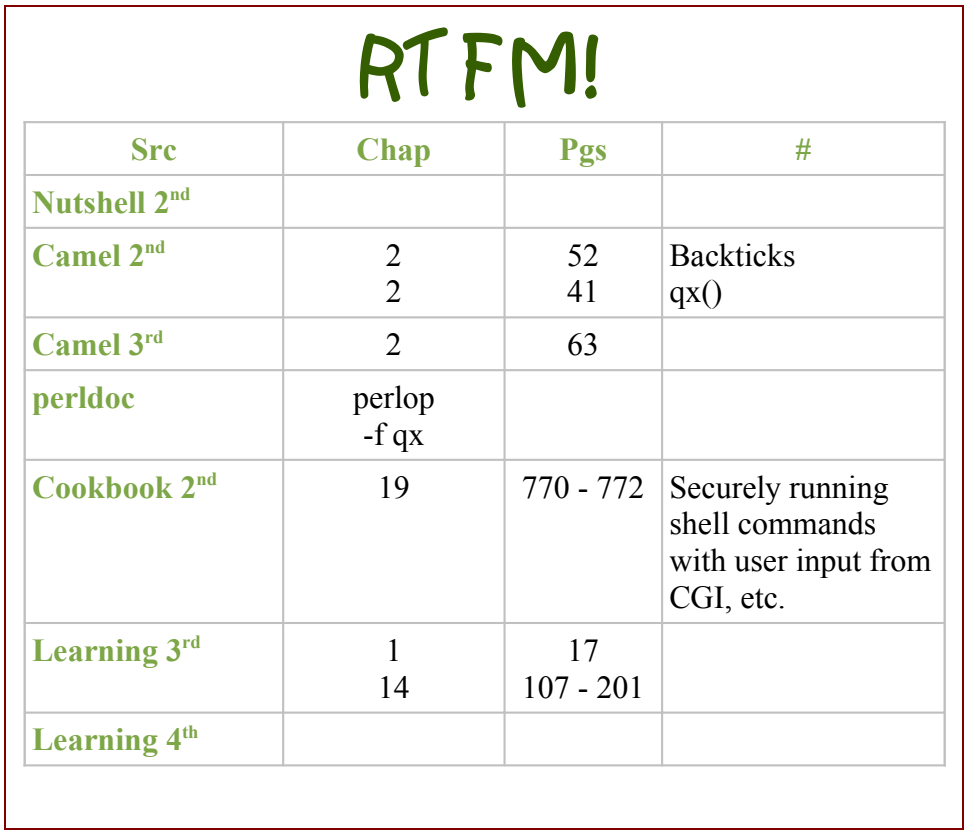
13.2 Using backticks

Single quotes can be used to specify a literal string which can be printed, assigned to a variable, et cetera. Double quotes perform interpolation of variables and certain escape sequences such as `\n` to create a string which can also be printed, assigned, etc.

A new set of quotes, called *backticks*, can be used to interpolate variables then run the resultant string as a shell command. The output of that command can then be printed, assigned, and so forth.

Backticks are the backwards-apostrophe character (`` ` `` which appears below the tilde (~), next to the number `1` on most keyboards.

Just as the `q()` and `qq()` functions can be used to emulate single and double quotes and save you from having to escape quotemarks that appear within a string, the equivalent function `qx()` can be used to emulate backticks.



Src	Chap	Pgs	#
Nutshell 2nd			
Camel 2nd	2 2	52 41	Backticks qx()
Camel 3rd	2	63	
perldoc	perlop -f qx		
Cookbook 2nd	19	770 - 772	Securely running shell commands with user input from CGI, etc.
Learning 3rd	1 14	17 107 - 201	
Learning 4th			

13.2.1 Exercises

1. Modify your earlier finger program to use backticks instead of `system()`

(Answer: `exercises/answers/backtickfinger.pl`)

2. Change it to use `qx()` instead (Answer: `exercises/answers/qxfinger.pl`)
3. The Unix command **whoami** gives your username. Since most shells support backticks, you can type **finger `whoami`** to finger yourself. Use shell backticks inside your `qx()` statement to do this from within your Perl program. (Answer: `exercises/answers/qxfinger2.pl`)

13.3 Platform dependency issues

Note that the examples given above will not work consistently on all operating systems. In particular, the use of `system()` calls or backticks with Unix-specific commands will not work under Windows NT, MacOS, etc. Slightly less obviously, the use of backticks on NT can sometimes fail when the output of a command is sent explicitly to the screen rather than being returned by the backtick operation.

13.4 Security considerations

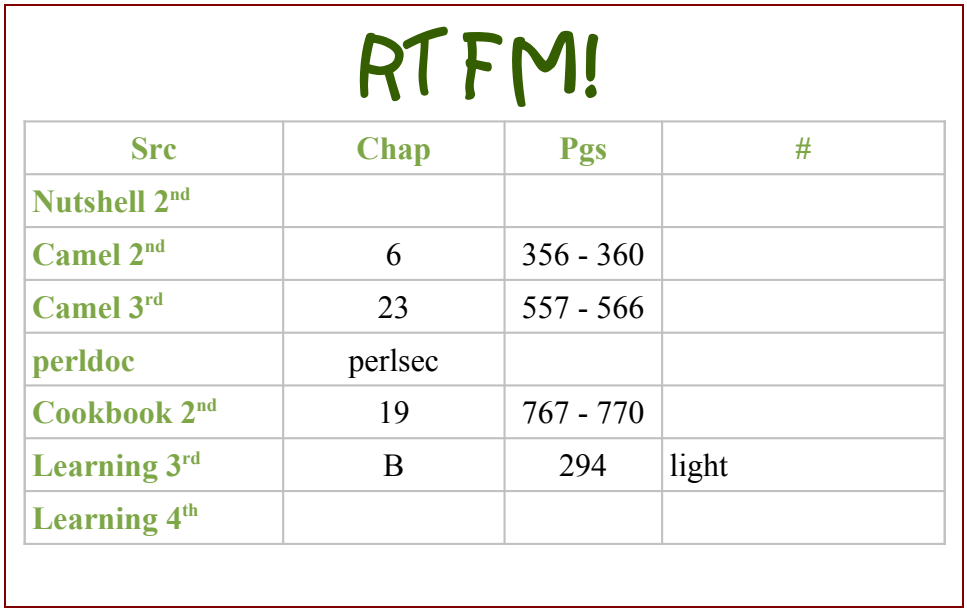
Many of the examples given above can result in major security risks if the commands executed are based on user input. Consider the example of a simple finger program which asked the user who they wanted to finger:

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

use strict;

print "Who do you want to finger? ";
my $username = <STDIN>;
print `finger $username`;
```

Imagine if the user's input had been `skud; cat /etc/passwd`, or worse yet, `skud; rm -rf /`. The system would perform both commands as though they had been entered into the shell one after the other.

Luckily, Perl's `-T` flag can be used to check for unsafe user inputs.

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



Src	Chap	Pgs	#
Nutshell 2nd			
Camel 2nd	6	356 - 360	
Camel 3rd	23	557 - 566	
perldoc	perlsec		
Cookbook 2nd	19	767 - 770	
Learning 3rd	B	294	light
Learning 4th			

`-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.

The only thing that will clear tainting is referencing substrings from a regexp match. The `perlsec` online documentation contains a simple example of how to do this. Read it now, and use it to complete the following exercises.

Note that you'll also have to explicitly set `$ENV{'PATH'}` to something safe (like `/bin`) as well.

13.4.1 Exercises

1. Modify the finger program above to perform taint checking (Answer: `exercises/answers/taintfinger.pl`)
2. Take one of your scripts using `open()` or `opendir()` and modify it to accept a filename as user input. Turn taint checking on. What sort of regular expression could you use to check for valid filenames? (Answer: `exercises/answers/taintfile.pl`)

13.5 Section 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 14: References and complex data structures

In this section...

In this section, 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.

14.1 Assumed knowledge

For this section, 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.

14.2 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.

Src	Chap	Pgs	#
Nutshell 2nd	4	75 - 77	
Camel 2nd	4	243 - 275	
Camel 3rd	8	242 - 267	
perldoc	perlref		
Cookbook 2nd	11	407 - 443	
Learning 3rd	B	296	light
Learning 4th			

Also Chapter 1 in *Advanced Perl Programming* and Tom Christiansen's FMTEYEWTK (Far More Than You Ever Wanted To Know) tutorials contain information about references. They're available from the Perl website (http://www.perl.com/)

14.3 Uses for references

There are three main uses for Perl references.

14.3.1 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.

14.3.2 Passing arrays and hashes to subroutines and functions

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

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

printargs(@a1, @a2);

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

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

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

14.3.3 Object oriented Perl

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

14.4 Creating and dereferencing references

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

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

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

Note that all references are scalars, because they contain a single item of information: the memory address of the actual data.

This is what a reference looks like if you print it out:

```
% perl -e 'my $foo_ref = \$foo; print "$foo_ref\n";'
SCALAR(0x80c697c)
% perl -e 'my $bar_ref = \@bar; print "$bar_ref\n";'
ARRAY(0x80c6988)
% perl -e 'my $baz_ref = \%baz; print "$baz_ref\n";'
HASH(0x80c6988)
```

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

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd	4 5	77 126	
Camel 2nd	3 4	204 251 - 252	Other tricks with references
Camel 3rd	8 29	258 773	
perldoc	-f ref		
Cookbook 2nd	11 13	409 499	
Learning 3rd			
Learning 4th			

Also in *Advanced Perl Programming*.

Dereferencing (getting at the actual data that a reference points to) is achieved by prepending the appropriate variable-type punctuation to the name of the reference. For instance, if we have a hash reference `$hash_reference` we can dereference it by looking for `%$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 `new_scalar`) you can put a reference variable (like `$scalar_ref`).

Here's how you access array elements or slices, and hash elements:

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

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];
print $hash_ref->{'sky'};
```

14.5 Passing multiple arrays/hashes as arguments

If we were attempt to pass two arrays together to a subroutine, they would be flattened out to form one large array.

```
my @fruits  = qw(apple orange pear banana);
my @rodents = qw(mouse rat hamster gerbil rabbit);
my @books   = qw(camel llama panther sheep);

mylist(@fruit, @rodents);

# print out all the fruits and then all the rodents
sub mylist {
        my @list = @_;
        foreach (@list) {
                print "$_\n";
        }
}
```

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

```
myreflist(\@fruit, \@rodents);

sub myreflist {
        my ($firstref, $secondref) = @_;
        print "First list:\n";
        foreach (@$firstref) {
                print "$_\n";
        }
        print "Second list:\n";
        foreach (@$secondref) {
                print "$_\n";
        }
}
```

14.6 Complex data structures

References are most often used to create complex data structures. Since hashes and arrays only accept scalars as elements, references (which are inherently scalars) can be used to create arrays of arrays or hashes, and hashes of arrays or hashes.

```
my %categories = (
        'fruits'        =>      \@fruits,
        'rodents'       =>      \@rodents,
        'books'         =>      \@books,
);

# to print out "gerbil"...
print $categories{'rodents'}->[3];
```

14.7 Anonymous data structures

We can use anonymous data structures to create complex data structures, to avoid having to declare many temporary variables. Anonymous arrays are created by using square brackets instead of round ones. Anonymous hashes use curly brackets instead of round ones.

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

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

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

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

my $array_ref = [qw(a b c d)];

# more useful yet, put these anon arrays straight into a hash:

my %transport = (
        'cars'          =>   [qw(toyota ford holden porsche)],
        'planes'        =>   [qw(boeing harrier)],
        'boats'         =>   [qw(clipper skiff dinghy)],
);
```

The same technique can be used to create anonymous hashes:

```
# The old, two-step way:

my %hash = (
        a       =>      1,
        b       =>      2,
        c       =>      3
```

```
);
my $hash_ref = \$hash;

# the quicker way, with an anonymous hash:
my $hash_ref = {
        a       =>      1,
        b       =>      2,
        c       =>      3
};
```

14.8 Exercises

1. Create a complex data structure as follows:
    a. Create a hash called `%pizza_prices` which contains prices for small, medium and large pizzas.
    b. Create a hash called `%pasta_prices` which contains prices for small, medium and large serves of pasta.
    c. Create a hash called `%milkshake_prices` which contains prices for small, medium and large milkshakes.
    d. Create a hash containing references to the above hashes, so that given a type of food and a size you can find the price of it.
    e. Convert the above hash to use anonymous data structures instead of the original three pizza, pasta and milkshake hashes
    f. Add a new element to your hash which contains the prices of salads

    (Answer: `exercises/answers/food.pl`)
2. 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. (Answer: `exercises/answers/exists.pl`)

14.9 Section summary

- References are scalar data consisting of the memory address of a piece of Perl data, and can be used in arrays, hashes, etc wherever 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.

Chapter 15: About databases

In this chapter...

This chapter talks about databases in general, and the different types of databases which can be used with Perl.

15.1 What is a database?

- A database is a collection of related information.
- The data stored in a database is persistent.

15.2 Types of databases

There are many different types of databases, including:

- Flat-file text databases
- Associative flat-file databases such as Berkeley DB
- Relational databases
- Object databases
- Network databases
- Hierarchical databases such as LDAP

Relational databases are by far the most useful type commonly available, and this training module focusses largely on them, after looking briefly at flat file text databases.

15.3 Database management systems

A database management system (DBMS) is a collection of software which can be used to create, maintain and work with databases. A client/server database system is one in which the database is stored and managed by a database server, and client software is used to request information from the server or to send commands to the server.

15.4 Uses of databases

Databases are commonly used to store bodies of data which are too large to be managed on paper or through simple spreadsheets. Most businesses use databases for accounts, inventory, personnel, and other record keeping. Databases are also becoming more widely used by home users for address books, cd collections, recipe archives, etc. There are very few fields in which databases cannot be used.

15.5 Chapter summary

- A database is a collection of related information.
- Data stored in a database is persistent
- There are a number of different types of databases, including flat file, relational, and others
- Database management systems are collections of software used to manage databases
- Databases are widely used in many fields

Chapter 16: Textfiles as databases

In this chapter...

In this chapter we investigate text-based or "flat file" databases and how to use Perl to manipulate them. We also discuss some of the limitations of this database format.

16.1 Delimited text files

A delimited text file is one in which each line of text is a record, and the fields are separated by a known character.

The character used to delimit the data varies according to the type of data. Common delimiters include the tab character (`\t` in Perl) or various punctuation characters. The delimiter should always be one which does not appear in the data.

Delimited text files are easily produced by most desktop spreadsheet and database applications (eg Microsoft Excel, Microsoft Access). You can usually choose "File" then "Save As" or "Export", then select the type of file you would like to save as.

Imagine a file which contains peoples' given names, surnames, and ages, delimited by the pipe (`|`) symbol:

```
Fred|Flintstone|40
Wilma|Flintstone|36
Barney|Rubble|38
Betty|Rubble|34
Homer|Simpson|45
Marge|Simpson|39
Bart|Simpson|11
Lisa|Simpson|9
```

The file above is available in your exercises directory as `delimited.txt`.

16.1.1 Reading delimited text files

To read from a delimited text file:

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

use strict;

open (INPUT, "delimited.txt") or die "Can't open data file: $!";

while (<INPUT>) {
        chomp;                  # remove newline
        my @fields = split(/\|/, $_);
```

```
        print "$fields[1], $fields[0]: $fields[2]\n";
}

close INPUT;
```

This should print out:

```
Flintstone, Fred: 40
Flintstone, Wilma: 36
...
```

And so on.

16.1.2 Searching for records

One of the common uses of databases is to search for specific records.

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

use strict;

# Find out what record the user wants:

print "Search for: ";
chomp (my $search_string = <STDIN>);

open (INPUT, "delimited.txt") or die "Can't open data file: $!";

while (<INPUT>) {
        chomp;                  # remove newline
        my @fields = split(/\|/, $_);

        # test whether the string matches given or family name
        if ($fields[0] =~ /$search_string/
                or $fields[1] =~ /$search_string/) {
                print "$fields[1], $fields[0]: $fields[2]\n";
        }
}

close INPUT;
```

16.1.3 Sorting records

Sorting records from a flat text database can be quite difficult. Simply sorting the items line by line is one simplistic approach:

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

use strict;

open (INPUT, "delimited.txt") or die "Can't open data file: $!";

my @records = sort <INPUT>;

foreach (@records) {
        chomp;                  # remove newline
        my @fields = split(/\|/, $_);
        print "$fields[1], $fields[0]: $fields[2]\n";
}

close INPUT;
```

The above technique can only sort on the first field of the data (in the case of our example, that would be the given name) and may have difficulties when it encounters the delimiter.

To sort by any other field, we would first need to load the data into a list of lists (using references), then use the `sort()` function's optional first argument to specify a subroutine to use for sorting:

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

use strict;

open (INPUT, "delimited.txt") or die "Can't open data file: $!";

while (<INPUT>) {
        chomp;
        my @this_record = split(/\|/, $_);

        # build a list-of-lists containing references to each record
        push (@records, \@this_record);
```

```
}

# sort takes an optional argument of what subroutine to use to sort
# the data...

my @sorted = sort given_name_order @records;

foreach $record (@sorted) {
        # we have to print the items via a reference to the array...
        print "$record->[1], $record->[0]: $record->[2]\n";
}

# subroutine to implement sorting order
sub given_name_order {
        $a->[0] cmp $b->[0];
}
```

Obviously this can be quite tricky, especially if the programmer is not totally familiar with Perl references. It also requires loading the entire data set into memory, which would be very inefficient for large databases.

16.1.4 Writing to delimited text files

The most useful function for writing to delimited text files is `join`, which is the logical equivalent of `split`.

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

use strict;

open OUTPUT, ">>delimited.txt" or die "Can't open output file: $!";

my @record = qw(George Jetson 35);

print OUTPUT join("|", @record), "\n";
```

16.2 Comma-separated variable (CSV) files

Comma separated variable files are another format commonly produced by spreadsheet and database programs. CSV files delimit their fields with commas, and wrap textual data in quotation marks, allowing the textual data to contain commas if required:

```
"Fred","Flintstone",40
"Wilma","Flintstone",36
"Barney","Rubble",38
"Betty","Rubble",34
"Homer","Simpson",45
"Marge","Simpson",39
"Bart","Simpson",11
"Lisa","Simpson",9
```

CSV files are harder to parse than ordinary delimited text files. The best way to parse them is to use the Text::ParseWords module:

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

use strict;
use Text::ParseWords;

open INPUT, "csv.txt" or die "Can't open input file: $!";

while (<INPUT>) {
        my @fields = quotewords("," 0, $_);
}
```

The three arguments to the `quotewords()` routine are:

- The delimiter to use
- Whether to keep any backslashes that appear in the data (zero for no, one for yes)
- A list of lines to parse (in our case, one line at a time)

16.3 Problems with flat file databases

16.3.1 Locking

When using flat file databases without locking, problems can occur if two or more people open the files at the same time. This can cause data to be lost or corrupted.

If you are implementing a flat file database, you will need to handle file locking using Perl's `flock` function.

16.3.2 Complex data

If your data is more complex than a single table of scalar items, managing your flat file database can become extremely tedious and difficult.

16.3.3 Efficiency

Flat file databases are very inefficient for large quantities of data. Searching, sorting, and other simple activities can take a very long time and use a great deal of memory and other system resources.

16.4 Chapter summary

- The two main types of text database use either delimited text or comma separated variables to store data
- Delimited text can be read using Perl's `split` function and written using the `join` function
- Comma separated files are most easily read using the Text::ParseWords module
- There are several problems with flat file databases including locking, efficiency, and difficulties in handling more complex data

Chapter 17: Relational databases

In this chapter...

The first section of this training session focuses on database theory, and covers relational database systems, and SQL - the language used to talk to them.

17.1 Tables and relationships

In a relational database, data is stored in tables. Each table contains data about a particular type of entity (either physical or conceptual).

For instance, our sample database is the inventory and sales system for Acme Widget Co. It has tables containing data for the following entities:

Table 4-1. Acme Widget Co Tables

Table	Description
stock_item	Inventory items
customer	Customer account details
saleperson	Sales people working for Acme Widget Co.
sales	Sales events which occur

Tables in a database contain fields and records. Each record describes one entity. Each field describes a single item of data for that entity. You can think of it like a spreadsheet, with the rows being the records and the columns being the fields, thus:

Table 4-2. Sample table

ID number	Description	Price	Quantity in stock
1	widget	$9.95	12
2	gadget	$3.27	20

Every table must have a *primary key*, which is a field which uniquely identifies the record. In the example above, the Stock ID number is the primary key.

The following figures show the tables used in our database, along with their field names and primary keys (in bold type).

Table 4-3. the stock_item table

stock_item
id
description
price
quantity

Table 4-4. the customer table

customer
id
name
address
suburb
state
postcode

Table 4-5. the salesperson table

salesperson
id
name

Table 4-6. the sales table

sales
id
sale_date
salesperson_id
customer_id
stock_item_id
quantity
price

17.2 Structured Query Language

SQL is a semi-English-like language used to manipulate relational databases. It is based on an ANSI standard, though very few SQL implementations actually adhere to the standard.

SQL statements are mostly case insensitive these days. While most books and references use upper-case, these notes use lower-case throughout for readability, and because the likelihood of needing to deal with older databases which only understand upper-case is becoming increasingly slim.

The syntax given in these course notes is cut down for simplicity; for full information, consult your database system's documentation. The MySQL documentation is available on our system in `/usr/doc/mysql-doc` and `/usr/doc/mysql-manual`, or by pointing your web browser at `http://training.netizen.com.au/`.

17.2.1 General syntax

SQL is case usually insensitive, apart from table and field names (which may or may not be case sensitive depending on what platform you're on -- on Unix they are usually case sensitive, on Windows they usually aren't).

String data can be delimited with either double or single quotes. Numerical data does not need to be delimited.

Wildcards may be used when searching for string data. A `%` (percent) sign is used to indicated multiple characters (much as an asterisk is used in DOS or Unix filename wildcards) while the underscore character (_) can be used to indicate a single character, similar to the `?` under Unix or DOS.

The following comparison operators may be used:

Table 4-7. Comparison Operators

Operator	Meaning
=	Equality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Inequality
like	Wildcard matching

In the following syntax examples, the term *condition* is used as shorthand for any expression which can be evaluated for truth, for instance `2 + 2 = 4` or `name like "A%"`.

Conditions may be combined by using `and` and `or`; use round brackets to indicate precedence. For instance, `name like "A%" or name like "B%"` will find all records where the ``name'' field starts with A or B.

17.2.1.1 SELECT

An SQL `select` statement is used to select certain rows from a table or tables. A select query will return as many rows as match the criteria.

17.2.1.1.1 Syntax

```
select field1 [, field2, field3] from table1 [, table2]
        where condition
        order by field [desc]
```

17.2.1.1.2 Examples

```
select id, name from customer;
select id, name from customer order by name;
select id, name from customer order by name desc;
```

We can use a select statement to obtain data from multiple tables. This is referred to as a ``join''.

```
select * from customer, sales where customer.id = sales.customer_id
```

17.2.1.2 INSERT

An `insert` query is used to add data to the database, a row at a time.

- The columns names are optional to make typing queries easier. This is fine for interactive use, however it is very bad practice to omit them in programs. *Always* specify column names in `insert` statements.

17.2.1.2.1 Syntax

```
insert into tablename (col_name1, col_name2, col_name3) values
(value1, value2, value3);
```

17.2.1.2.2 Examples

```
insert into stock_item (id, description, price, quantity) values (0,
'doodad', 9.95, 12);
```

Note that since the `id` field is an `auto_increment` field in the Acme inventory database we've set up, we don't need to specify a value to go in there, and just use zero instead --- whatever we specify will be replaced with the auto-incremented value. Auto-increment fields of some kind are available in most database systems, and are very useful for creating unique ID numbers.

17.2.1.3 DELETE

A `delete` query can be used to delete rows which match a given criteria.

17.2.1.3.1 Syntax

```
delete from tablename where condition
```

17.2.1.3.2 Examples

```
delete from stock_item where quantity = 0;
```

17.2.1.4 UPDATE

The `update` query is used to change the values of certain fields in existing records.

17.2.1.4.1 Syntax

```
update tablename set field1 = expression, field2 = expression
        where condition
```

17.2.1.4.2 Examples

```
update stock_item set quantity = (quantity - 1) where id = 4;
```

17.2.1.5 CREATE

The `create` statement is used to create new tables in the database.

17.2.1.5.1 Syntax

```
create table tablename (
        column coltype options,
```

```
        column coltype options,
        ...
        primary key (colname)
)
```

Data types include (but are not limited to):

Table 4-8. Some data types

INT	an integer number
FLOAT	a floating point number
CHAR(*n*)	character data of exactly *n* characters
VARCHAR(*n*)	character data of up to *n* characters (field grows/shrinks to fit)
BLOB	Binary Large OBject
DATE	A date in YYYY-MM-DD format
ENUM	enumerated string value (eg "Male" or "Female")

Data types vary slightly between different database systems. The full range of MySQL data types is outlined in section 7.2 of the MySQL reference manual.

17.2.1.5.2 Examples

```
create table contactlist (
        id int not null auto_increment,
        name varchar(30),
        phone varchar(30),
        primary key (id)
)
```

17.2.1.6 DROP

The `drop` statement is used to delete a table from the database.

17.2.1.6.1 Syntax

```
drop table tablename
```

17.2.1.6.2 Example

```
drop table contactlist
```

17.3 Chapter summary

- A database table contains fields and records of data about one entity
- SQL (Structured Query Language) can be used to manipulate and retrieve data in a database
- A `SELECT` query may be used to retrieve records which match certain criteria
- An `INSERT` query may be used to add new records to the database
- A `DELETE` query may be used to delete records from the database
- An `UPDATE` query may be used to modify records in the database
- A `CREATE` query may be used to create new tables in the database
- A `DROP` query may be used to remove tables from the database

Chapter 18: MySQL

In this chapter...

In this section we examine the popular database MySQL, which is available for free for many platforms. MySQL is just one of many database systems which can be accessed via Perl's DBI module.

18.1 MySQL features

18.1.1 General features

- Fast
- Lightweight
- Command-line and GUI tools
- Supports a fairly large subset of SQL, including indexing, binary objects (BLOBs), etc
- Allows changes to structure of tables while running
- Wide userbase
- Support contracts available

18.1.2 Cross-platform compatibility

- Available for most Unix platforms
- Available for Windows NT/95/98 (there are license differences)
- Available for OS/2
- Programming libraries for C, Perl, Python, PHP, Java, Delphi, Tcl, Guile (a scheme interpreter), and probably more...
- Open-source ODBC

18.2 Comparisions with other popular DBMSs

18.2.1 PostgreSQL

MySQL and PostgreSQL are very similar in many ways. MySQL is driven by one company while PostgreSQL is an open source project with major contributions coming from a variety of companies and individuals.

More information: `http://www.postgresql.org/`

18.2.2 Oracle, Sybase, etc

MySQL will not give you the features of Oracle or other enterprise-level database management systems. In particular, MySQL lacks triggers and views. The price you pay for this is that Oracle costs a lot, and requires heavy hardware to run on and is much more maintenance intensive. MySQL is better suited to small-to-medium database applications such as web-based database applications, and will do so happily on a common PC.

More information: `http://www.oracle.com/`

18.3 Getting MySQL

MySQL can be downloaded from `http://www.mysql.com/`or mirror sites worldwide. It is also available in packaged binary format for various operating system distributions, including RedHat and Debian linux.

Installation instructions come with the software, but in brief:

18.3.1 Redhat Linux

Download the appropriate RPM packages, and type `rpm -i` *`packagename`*`.rpm`

18.3.2 Debian Linux

Use `apt-get`, `dselect`, or `dpkg` to install the `.deb` packages. For instance, `apt-get install mysql`.

18.3.3 Compiling from source

Download the `tar.gz` file from `http://www.mysql.com/` and read the `README` file. Then type `./configure`, `make`, and `make install`.

18.3.4 Binaries for other platforms

Binaries are available for many platforms, including Windows and some commercial Unix platforms. Follow the installation instructions found in the `README` file.

18.4 Setting up MySQL databases

A tool called `mysqladmin` is distributed with MySQL. This tool allows the database administrator (DBA) to create, remove, or otherwise manage databases.

Table 5-1. Mysqladmin commands:

create *databasename*	Create a new database
drop *databasename*	Delete a database and all its tables
flush-hosts	Flush all cached hosts
flush-logs	Flush all logs
flush-tables	Flush all tables
kill *id,id,...*	Kill mysql threads
password *new-password*	Change old password to new-password
processlist	Show list of active threads in server
reload	Reload grant tables
refresh	Flush all tables and close and open logfiles
shutdown	Take server down
status	Gives a short status message from the server
variables	Prints variables available
version	Get version info from server

More help for this command is available by typing `mysqladmin --help` from the command line or by reading the MySQL reference manual.

18.4.1 Creating the Acme inventory database

To create a database called `inventory`, we would perform the following steps as the user who has permission to run `mysqladmin` (eg root):

```
% mysqladmin create inventory
% mysqladmin reload
```

18.4.2 Setting up permissions

To set up security permissions for the inventory database, we would need to create appropriate records in the `mysql` database (that's right, it's a database which has the same name as the database server). This is the central repository for access control information for all databases served by your MySQL server.

Typically, you will want to:

- create an entry in the `db` table for the database
- set the default permissions for the database
- create an entry in the `user` table for any users who should be allowed to access the database
- set default permissions for each user

All these are achieved by performing simple INSERT or UPDATE queries on the tables in question.

Table 5-2. Available permissions include ...

Select	May perform SELECT queries
Insert	May perform INSERT queries
Update	May perform UPDATE queries
Delete	May perform DELETE queries
Create	May create new tables
Drop	May drop (delete) tables
Reload	May reload the database
Shutdown	May shut down the database
Process	Has access to processes on the OS
File	Has access to files on the OS's file system

18.4.3 Creating tables

The SQL statements used to create tables are documented in the MySQL manual. `CREATE` statements are used to create each individual table by specifying the fields for each table, their data types and other options.

Below is an example --- these SQL statements create the Acme Widget Co. tables we will be working with throughout this session. The output you see is generated by the **mysqldump** program, and can be read back into a database via command line redirection, eg **mysql *database* < *filename***.

```
#
# Table structure for table 'customer'
#
CREATE TABLE customer (
  id int(11) DEFAULT '0' NOT NULL auto_increment,
  name varchar(80),
  address varchar(255),
  suburb varchar(50),
  state char(3),
  postcode char(10),
  PRIMARY KEY (id)
);

#
# Table structure for table 'sales'
#
CREATE TABLE sales (
  id int(11) DEFAULT '0' NOT NULL auto_increment,
  sale_date date,
  customer_id int(11),
  salesperson_id int(11),
  stock_item_id int(11),
  quantity int(11),
  price float(4,2),
  PRIMARY KEY (id)
);

#
# Table structure for table 'salesperson'
#
CREATE TABLE salesperson (
  id int(11) DEFAULT '0' NOT NULL auto_increment,
  name varchar(80),
  PRIMARY KEY (id)
);

#
# Table structure for table 'stock_item'
#
CREATE TABLE stock_item (
  id int(11) DEFAULT '0' NOT NULL auto_increment,
```

```
  description varchar(80),
  price float(4,2),
  quantity int(11),
  PRIMARY KEY (id)
);
```

18.5 The MySQL client

To talk to any database server, you will need to use a client of some kind. MySQL comes with a text-based client by default, but there are graphical clients available, as well as ODBC drivers to allow you to interact with a MySQL database from Windows applications such as Microsoft Access.

The command line client can be invoked from the command line with the `mysql` command. The `mysql` command takes a database name as a required argument, as well as other optional arguments such as `-p`, which causes the client to ask for a password for access to the database if access controls have been set up.

You can see all the options available on the command line by typing `mysql -help`.

> **ADVANCED**
>
> You can set up access controls on a database by editing the data in the `mysql` database (i.e. type `mysql mysql` on the command line) or by using the `mysqlaccess` command. Type `mysqlaccess --help` for more information about this command.

```
$ mysql -p databasename
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 3.21.33b

Type 'help' for help.

mysql>
```

The MySQL client allows you to type in commands on one or many lines. When you finish a statement, type `;` to end, same as for Perl.

To quit the client, type `quit` or `\q`.

For a full outline of commands available in the client, type `help` or `\h`. It will give you this message:

```
mysql> \h
```

```
MySQL commands:
help    (\h)    Display this text
?       (\h)    Synonym for `help'
clear   (\c)    Clear command
connect (\r)    Reconnect to the server. Optional arguments are db
and host
edit    (\e)    Edit command with $EDITOR
exit    (\)     Exit mysql. Same as quit
go      (\g)    Send command to mysql server
print   (\p)    print current command
quit    (\q)    Quit mysql
rehash  (\#)    Rebuild completion hash
status  (\s)    Get status information from the server
use     (\u)    Use another database. Takes database name as
argument

Connection id: 1  (Can be used with mysqladmin kill)
```

18.6 Understanding the MySQL client prompts

The prompt that shows when you are using the MySQL client tells you a lot about what's going on.

The normal prompt looks like this:

```
mysql>
```

This means it is waiting for you to enter an SQL query.

If you are in the middle of entering an SQL query, it will be waiting for a semi-colon to terminate the query, and will look like this:

```
    ->
```

If you have opened a set of quotes but not closed them, you will see one of these prompts:

```
    '>
    ">
```

18.7 Exercises

1. Connect to a database which has the same name as your login (for instance, `train01`) by typing **mysql -p train01** (the `-p` flag causes it to ask you for your password, which in this case is the same as your login password). The database you are connecting to is your own personal copy of the Acme Widget Co. inventory and sales database mentioned in the previous section
2. Type `show tables` to show a list of tables in this database
3. Type `describe customer` to see a description of the fields in the table `customer`
4. Type `select * from customer` to perform a simple SQL query
5. Try selecting fields from other tables. Try both `select *` and `select field1, field2` type queries.
6. Use the `where` clause to limit which records you select
7. Use the `order by` clause to change the order in which records are returned
8. Insert a record into the customer table which contains your own name and address details
9. Update the price of widgets in the `stock_item` table to change their price to $19.95

When developing database applications, it is often useful to keep a client program such as this one open to test queries or check the state of your data. You can open multiple telnet sessions to our training system to do this if you wish.

18.8 Chapter summary

- MySQL is one of many database systems which can be used as the back-end to a web site
- MySQL can be downloaded from http://www.mysql.com/ or mirror sites
- The MySQL command line client can be used to interact with MySQL databases
- The MySQL client allows the user to type in SQL queries and prints results to the screen.

Chapter 19: The DBI and DBD modules

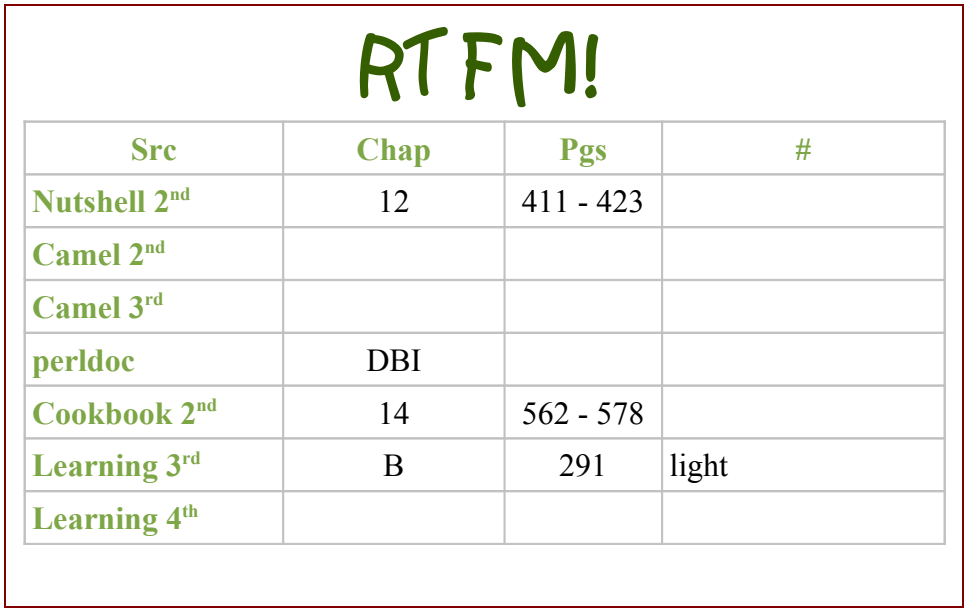
In this chapter...

In this section we look at the Perl module which can be used to interact with many database servers: DBI.

19.1 What is DBI?

Like the Perl modules discussed in last week's CGI programming course, the DBI and DBD modules are written by Perl people and distributed free via CPAN (the Comprehensive Perl Archive Network).

DBI stands for "Database Interface" while DBD stands for "Database Driver". You need both types of modules, working together, in order to access databases using Perl.



Src	Chap	Pgs	#
Nutshell 2nd	12	411 - 423	
Camel 2nd			
Camel 3rd			
perldoc	DBI		
Cookbook 2nd	14	562 - 578	
Learning 3rd	B	291	light
Learning 4th			

19.2 Supported database types

Databases supported by Perl's DBI module include:

- Oracle
- Sybase
- Informix
- MySQL
- Msql
- Ingres
- Postgres
- Xbase
- DB2
- ... and more

19.3 How does DBI work?

DBI is a generic interface which acts as a "funnel" between the programmer and multiple databases.

DBI protects you from needing to know the minutiae of connecting to different databases by providing a consistent interface for the programmer. The only thing you need to vary is the connection string, to indicate what sort of database you wish to connect to.

To use DBI, you need to install the DBI module from CPAN, as well as any DBD modules for the databases you use. For instance, to use MySQL you need to install the `DBD::Mysql` module.

> ADVANCED
>
> To install DBI, download the DBI module from CPAN (http://www.perl.com/CPAN), unzip it using a command like **tar -xzvf DBI.tar.gz**, then follow the instructions in the **README** file distributed with the module.

19.4 DBI/DBD syntax

The syntax of the database modules is best found by using the **perldoc** command. **perldoc DBI** will give you general information applicable to all DBI scripts, while **perldoc DBD::*yourdatabase*** will give information specific to your own database. In our case, we use **perldoc DBD::mysql**.

DBI is an object oriented Perl module, like the `Text::Template` and `Mail::Mailer` modules covered in the CGI Programming in Perl training module. This means that when we connect to the database we will be creating an object which is called a "database handle" which refers to a specific session with the database. Thus we can have multiple sessions open at once by creating multiple database handles.

We can also create statement handle objects, which are Perl objects which refer to a previously prepared SQL statement. Once we have a statement handle, we can use it to execute the underlying SQL as often as we want.

19.4.1 Variable name conventions

The following variable name conventions are used in the DBD/DBI documentation:

Table 6-1. DBI module variable naming conventions

Variable name	Meaning
`$dbh`	database handle object
`$sth`	statement handle object
`$rc`	Return code (boolean: true=ok, false=error)
`$rv`	Return value (usually an integer)
`@ary`	List of values returned from the database, typically a row of data
`$rows`	Number of rows processed (if available, else -1)

19.5 Connecting to the database

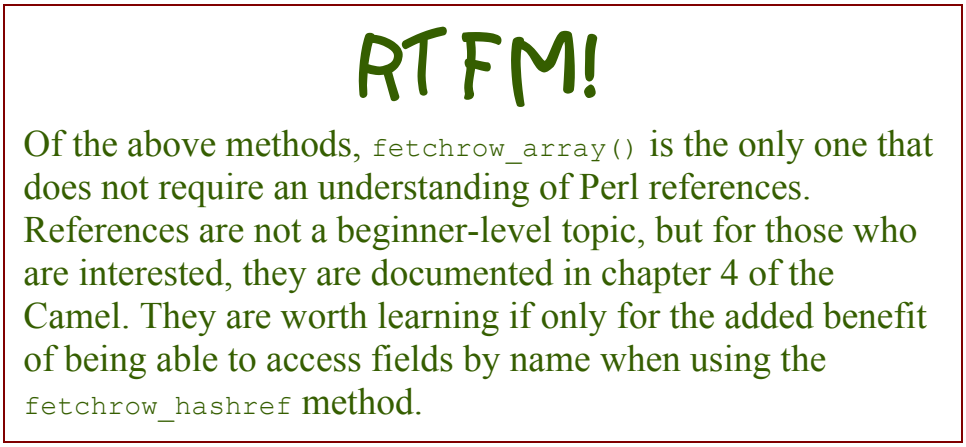
```
use DBI;

my $driver = 'mysql';
my $database = 'database_name';          # name of your database here
my $username = undef;                    # your database username
my $password = undef;                    # your database password

# note that username and password should be assigned to if your
database
# uses authentication (ie requires you to log in)

# we set up a connection string specific to this database
my $dsn = "DBI:$driver:database=$database";

# make the actual connection - this returns a database handle we can
use later
my $dbh = DBI->connect($dsn, $username, $password);

# when you're done (at the end of your script)
$dbh->disconnect();
```

19.6 Executing an SQL query

```
# set up an SQL statement
my $sql_statement = "select * from customer";
my $sth = $dbh->prepare($sql_statement)
        || die "Could not prepare: " . $dbh->errstr();

# execute it
$sth->execute() || die "Could not execute: " . $dbh->errstr();

# how many rows did we get?
my $num_rows = $sth->rows();
my $num_fields = $sth->{'NUM_OF_FIELDS'};

# close the sql query, if we don't want it any more.
$sth->finish();
```

19.7 Doing useful things with the data

```
# get an array full of the next row of data that matches the query
# (the most common, and simplest, case)
while (my @ary = $sth->fetchrow_array()) {
        print "The first field is $ary[0]\n";
}

# get a hash reference instead
# (the more complicated, but more useful, version)
while (my $hashref= $sth->fetchrow_hashref()) {
        print "Name is $hashref->{'name'}\n";
}

# you can also get an arrayref
# (equally complicated and not quite as useful)
while (my $ary_ref = $sth->fetchrow_arrayref()) {
        print "The first field is $ary_ref->[0]\n";
}
```



19.8 An easier way to execute non-SELECT queries

If you wish to execute a query such as INSERT, UPDATE, or DELETE, you may find it easier to use the `do()` method:

```
$dbh->do("delete from sales")
    || warn("Can't delete from sales table");
```

This method returns the number of rows affected, or undef if there is an error.

19.9 Quoting special characters in SQL

Sometimes you want to use a value in your SQL which may contain characters which have special behaviour in SQL, such as a percent sign or a quote mark. Luckily, there is a method which can automatically escape all special characters:

```
my $string = "20% off all stock";
my $clean_string = $dbh->quote($string);
```

19.10 Exercises

1. Use `exercises/scripts/easyconnect.pl` to connect to your Acme Widget Co. database. You will need to edit some of the lines at the top.
2. Use a `while` loop to output data a row at a time
3. Check all your statements for indications of failure, and output messages to the user using `warn()` if any of the steps fail.

19.10.1 Advanced exercises

1. If you wish, you can use a hash reference instead of an array
2. Change the SQL in `easyconnect.pl` to use a non-SELECT statement, and use the `do` method instead of the `prepare` and `execute` methods. Don't forget to check the return value!

19.11 Chapter summary

- The DBI module provides a consistent interface to a variety of database systems
- The DBI module can be downloaded from CPAN
- Documentation for the DBI module can be found by typing **perldoc DBI**

Chapter 20: Acme Widget Co. Exercises

In this chapter...

In the second half of this training module, we will be tying together what we have learned about SQL and DBI, and creating a simple application for Acme Widget Co. to assist them in inventory management, sales, and billing.

20.1 The Acme inventory application

In your `exercises/` directory you will find a subdirectory called `acme/` which contains the outline of the Acme inventory application which you will build upon for the rest of today.

20.2 Listing stock items

The shell of a stock-listing script is available in your `exercises/acme/` directory as `stocklist.pl`.

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

my $driver = 'mysql';
my $database = 'trainXX';
my $username = 'trainXX';
my $password = 'your_password_here';

my $dsn = "DBI:$driver:database=$database";
my $dbh = DBI->connect($dsn, $username, $password)
        || die $DBI::errstr;

my $sql_statement = "select * from stock_item";
my $sth = $dbh->prepare($sql_statement);
$sth->execute() or die ("Can't execute SQL: " . $dbh->errstr());

while (my @ary = $sth->fetchrow_array()) {
        print <<"END";
ID:             $ary[0]
Description:    $ary[1]
Price:          $ary[2]
Quantity:       $ary[3]
END
}

$dbh->disconnect();
```

1. Fill in the variables indicated (`$database`, `$sql_statement`, etc)
2. Test your script from the command line
3. Sort the output in alphabetical order by Description

20.2.1 Advanced exercises:

1. If you are familiar with Perl references, convert the script to use `fetchrow_hashref()`
2. Ask the user to specify a field to sort by, either as a command line argument or on STDIN. If the sort order parameter is given, use it to change the sort order in your SQL statement and re-output the result, otherwise default to something sensible such as ID

20.3 Adding new stock items

1. Write a script which prompts the user for input, asking for values for description, quantity and price. Remember that the stock item's ID will be automatically filled in by the database, as it is an "auto increment" field.
2. Next, create an SQL query to add a record to the database. Output a message to the user indicating the success (or failure) of the operation. A sample script to get you started is available in `exercises/acme/addstock.pl`

20.3.1 Advanced exercises

1. Check that the price is a number (use regular expressions for these checks)
2. Check that it has two decimal places
3. Check that the number of items in stock is a number

20.4 Entering a sale into the system

1. The program `exercises/acme/sale.pl` provides an interface which can be used to input data pertinent to the occurence of a sale
2. Write a script which records the sale in the `sales` table
3. Your script will also have to update the `stock_item` table to reduce the number of items still in stock.
4. What happens if you try to buy/sell more items than are available? Put in a check to stop this from happening.

20.5 Creating sales reports

1. Copy the code from the previous example's script to create a script that asks the user for a salesperson's ID number and a start and end date.
2. Use the script to output a sales report for the chosen salesperson for the period between the two dates.

20.5.1 Advanced exercises

1. Create an extra option for "all" sales people, which shows all the sales people in descending order of sales made. You may need to use an SQL `group by` clause to achieve this.

20.6 Searching for stock items

1. Create a script which asks a user for a string to search for in a stock item's description (eg "dynamite").
2. Allow the user to choose either "Full name", "Beginning of name" or "Part of name" as a search type.
3. Create different SQL queries using `LIKE` to search the data depending on their choices

20.6.1 Advanced exercises

1. Change the script so that people can use DOS/Unix style wildcards (`*` and `?`) then use their wildcard expression in your SQL query - convert the wildcards to SQL-style wildcards by using regular expressions

Chapter 21: References

In this chapter...

This section is included as an optional topic. It is intended for those who have experience in C or other languages which use pointers and references.

RTFM!

References are covered at length in the first chapter of the O'Reilly book *Advanced Perl Programming* by Sriram Srinivasan (the "Panther" book). Lastly, **perldoc perlref** contains online documentation related to Perl references.

Uses for Perl references:

- creating complex data structures, for example multi-dimensional arrays
- passing multiple arrays and hashes to subroutines and functions without them getting smushed together
- creating anonymous data structures

21.1 Creating and deferencing

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.

Dereferencing (getting at the value that a reference points to) is achieved by prepending the appropriate variable-type punctuation to the name of the reference. For instance, if we have a hash reference `$hash_reference` we can dereference it by looking for `%$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 `new_scalar`) you can put a reference variable (like `$scalar_ref`).

Here's how you access array elements or slices, and hash elements:

```
print $$array_ref[0];         # prints the first element of the
                              # array referenced by $array_ref
print $$array_ref[0..2];      # prints an array slice
print $$hash_ref{'sky'};      # prints a hash element's value
```

The other way to access the value that a reference points to is using 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];
print $hash_ref->{'sky'};
```

The Panther book describes a good way to visualise this method. Ask your instructor to demonstrate it or to loan you a copy of the book if you need a better understanding of the above syntax.

21.2 Complex data structures

We can use references to create complex data structures, such as this hash in which the values are arrays rather than scalars. Actually, they are scalars, since the array references are scalars, but they point to arrays.

```
my @fruits = qw(apple orange pear banana);
my @rodents = qw(mouse rat hamster gerbil rabbit);
my @books = qw(camel llama panther);

my %categories = (
        'fruits'        =>              \@fruits,
        'rodents'       =>              \@rodents,
        'books'         =>              \@books,
);

# to print out "gerbil"...
print $categories->{'rodents'}->[3];
```

21.3 Passing multiple arrays/hashes as arguments

If we were to attempt to pass two arrays together to a function or subroutine, they would be flattened out to form one large array:

```
mylist(@fruit, @rodents);

# print out all the fruits then all the rodents
sub mylist {
        my @list = @_;
        foreach (@list) {
                print "$_\n";
        }
}
```

If we want them kept separate, pass references:

```
myreflist(@fruit, @rodents);

sub myreflist {
        my ($firstref, $secondref) = @_;
        print "First list:\n";
        foreach (@$firstref) {
                print "$_\n";
        }
        print "Second list:\n";
        foreach (@$secondref) {
                print "$_\n";
        }
}
```

21.4 Anonymous data structures

Lastly, references can be used to create anonymous data structures which are destroyed once you're done with them. An anonymous array is created by using square brackets instead of round ones. An anonymous hash uses curly brackets 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:

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

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

my $array_ref = [qw(a b c d)];

# more useful yet:

my %transport = (
        'cars'          =>      [qw(toyota ford holden porsche)],
        'planes'        =>      [qw(boeing harrier)],
        'boats'         =>      [qw(clipper skiff dinghy)],
);
```

21.5 Chapter summary

- References may be used to create complex data structures, pass multiple arrays and hashes to subroutines, and to create anonymous data structures
- References are created by prefixing the name of a variable with a backslash
- References are dereferenced by using the name of a reference (including the dollar sign) where we would usually use the alphanumeric name of a variable, or by using the arrow notation.
- References can be included in Perl data structures anywhere you might ordinarily find scalars.
- References to anonymous arrays may be created by initialising an array using square brackets instead of round ones.
- References to anonymous hashes may be created by initialising an hash using curly brackets instead of round ones.

Chapter 22: What is CGI?

In this chapter...

In this section we will define the term CGI and learn how web servers use CGI to provide dynamic and interactive material. We explore the Hypertext Transfer Protocol as it applies to both static and CGI-generated content, and examine raw HTTP requests and responses by telnetting to a web server.

22.1 Definition of CGI

CGI is the Common Gateway Interface, a standard for programs to interface with information servers such as HTTP (web) servers. CGI allows the HTTP server to run an executable program or script in response to a user request, and generate output on the fly. This allows web developers to create dynamic and interactive web pages.

CGI programs can be written in any language. Perl is a very common language for CGI programming as it is largely platform independent and the language's features make it very easy to write powerful applications. However, some CGI programs are written in C, shell script, or other languages.

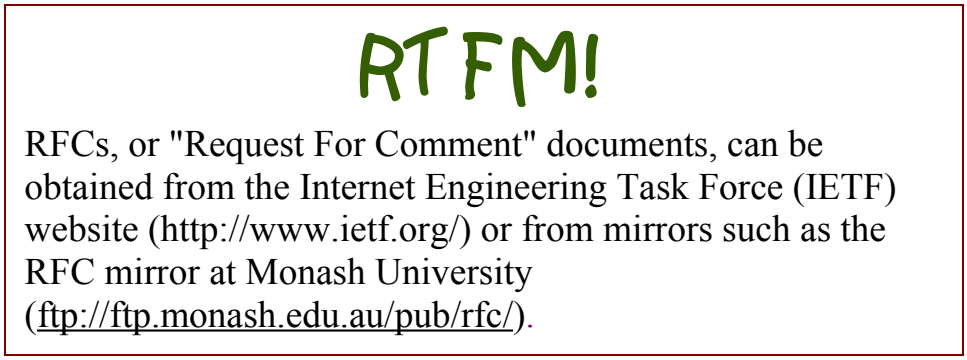
It is important to remember that CGI is not a language in itself. CGI is merely a type of program which can be written in any language.

22.2 Introduction to HTTP

To understand how CGI works, you need some understanding of how HTTP works.

HTTP stands for HyperText Transfer Protocol, and (not very surprisingly) is the protocol used for transferring hypertext documents such as HTML pages on the World Wide Web.

For the purposes of this course, we will only be looking at HTTP version The current version, 1.1, is specified in RFC 2068 and contains many more features, but none of them are necessary for a basic understanding of CGI programming. An HTTP cheat-sheet, containing some common terminology and a table of status codes, appears in Appendix E.



RFCs, or "Request For Comment" documents, can be obtained from the Internet Engineering Task Force (IETF) website (http://www.ietf.org/) or from mirrors such as the RFC mirror at Monash University (ftp://ftp.monash.edu.au/pub/rfc/).

A simple HTTP transaction, such as a request for a static HTML page, works as follows:

1. The user types a URL into his or her browser, or specifies a web address by some other means such as clicking on a link, choosing a bookmark, etc
2. The user agent connects to port 80 of the HTTP server
3. The user agent sends a request such as `GET /index.html`
4. The user agent may also send other headers
5. The HTTP server receives the request and finds the requested file in its filesystem
6. The HTTP server sends back some HTTP headers, followed by the contents of the requested file
7. The HTTP server closes the connection

When a user requests a CGI program, however, the process changes slightly:

1. The user agent sends a request as above

2. The HTTP server receives the request as above
3. The HTTP server finds the requested CGI program in its file system
4. The HTTP server executes the program
5. The program produces output
6. The output includes HTTP headers
7. The HTTP server sends back the output of the program
8. The HTTP server closes the connection

22.3 Terminology

authentication

The process by which a client sends username and password information to the server, in an attempt to become authorized to view a restricted resource.

client

An application program that establishes connections for the purpose of sending requests.

Content-type

The media type of the body of the response, as given in the `Content-type:` header. Examples include `text/html`, `text/plain`, `image/gif`, etc.

method

Indicates what the server should do with a resource. Case sensitive. Valid methods include: GET, HEAD, POST

request

An HTTP request message sent by a client to a server

resource

A network data object or service which can be identified by a URI.

response

An HTTP response message sent by a server to a client

server

An application program that accepts connections in order to service requests by sending back responses.

status code

A 3-digit integer indicating the result of the server's attempt to understand and satisfy the request. A table of status codes and their meanings appears below.

Uniform Resource Identifier (URI)

URIs are formatted strings which identify - via name, location, or any other characteristic - a network resource.

Uniform Resource Locator (URL)

: A web address. May be expressed absolutely (eg `http://www.example.com/services/index.html`) or in relation to a base URI (eg `../index.html`) See also URI.

user agent

: The client which initiates a request. These are often browsers, editors, spiders (web-traversing robots) or other end-user tools.

22.4 HTTP status codes

Table 2-1. HTTP status codes

Code	Meaning
200	OK
201	Created
202	Accepted
204	No Content
301	Moved Permanently
302	Moved Temporarily
304	Not Modified
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable

22.5 HTTP Methods

22.5.1.1 GET

The GET method means retrieve whatever information is identified by the request URI. If the request URI refers to a data-producing process (eg a CGI program), it is the produced data which is returned, and not the source text of the process.

22.5.1.2 HEAD

The HEAD method is identical to GET except that the server will only return the headers, not the body of the resource. The meta-information contained in the HTTP headers in response to a HEAD request should be identical to the information sent in response to a GET request. This method can be used to obtain meta-information about the resource without transferring the body itself.

22.5.1.3 POST

The POST method is used to request that the server use the information encoded in the request URI and use it to modify a resource such as:

- Annotation of an existing resource
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles
- Providing data {such as the result of submitting a form} to a data-handling process
- Updating a database

22.6 Exercises

The HTTP request/response process is usually transparent to the user. To see what's going on, let's connect directly to the web server and see what happens.

Login to the system as for the Introduction to Perl course:

1. Open the telnet program, TeraTerm
2. Connect to the training server (your instructor will give you the hostname or IP number)
3. Login using the username and password you were given
4. From the Unix command line, type **telnet localhost 80** -- this connects to port 80 of the server, where the HTTP daemon (aka the web server) is listening. You should see something like this:

```
training:~> telnet localhost 80
Trying 1.2.3.4
Connected to training.netizen.com.au.
Escape character is '^]'.
```

5. Ask the web server for a static document by typing: `GET /index.html HTTP/1.0` then press enter twice to send the request. Note that this command is *case sensitive*.
6. Look at the response that comes back. Do you see the headers? They should look something like this:

```
HTTP/1.1 200 OK
Date: Tue, 28 Mar 2000 02:42:37 GMT
Server: Apache/1.3.6 (Unix)
Connection: close
Content-Type: text/html
```

   This will be followed by a blank line, then the content of the file you asked for. Then you will see "Connection closed by foreign host", indicating that the HTTP server has closed the connection.

   - If you miss seeing the headers because the body is too long, try using the `HEAD` method instead of `GET`.

7. Telnet to port 80 again and ask the web server for a CGI script's output by typing `GET /cgi-bin/localtime.cgi HTTP/1.0`
8. Now let's get some status codes other than `200 OK` from the web server:

- `GET /not_here.html HTTP/1.0` (a file which doesn't exist)
- `GET /unreadable.html HTTP/1.0` (a file with the permissions set wrong)
- `GET /protected.html HTTP/1.0` (a file protected by HTTP authentication - we cover this later on today)
- `GET /redirected.html HTTP/1.0` (a file which is redirected to a different URL)
- `ENCRYPT /index.html HTTP/1.0` (a method which isn't known to our server)

22.7 What is needed to run CGI programs?

There are several things you need in order to create and run Perl CGI programs.

- a web server
- web server configuration which gives you permission to run CGI
- a Perl interpreter
- appropriate Perl modules, such as CGI.pm
- a shell account is extremely useful but not essential

Most of the above requirements will need your system administrator or ISP to set them up for you. Some will be wary of allowing users to run CGI programs, and may require you to obey certain security regulations or pay extra for the privilege. The most common security requirement is that CGI programs must run under cgiwrap. This is discussed later, in the section on security.

22.8 Chapter summary

- CGI stands for Common Gateway Interface
- HTTP stands for Hypertext Transfer Protocol. This is the protocol used for transferring documents and other files via the World Wide Web.
- HTTP clients (web browsers) send requests to HTTP (web) servers, which are answered with HTTP responses
- The request/response can be examined by telnetting to the appropriate port of a web server and typing in requests by hand.

Chapter 23: Generating web pages with Perl

In this chapter...

In this section, we will create a simple "Hello world" CGI program and run it, then extend upon that to integrate parts of Perl taught in previous modules. Alternative quoting mechanisms are briefly covered, and we also discuss debugging techniques for CGI programs.

23.1 Your public_html directory

The training server has been set up so that each user has their own web space underneath their home directory. All files which will be accessible via the web should be placed in the directory named `public_html`. This is common for most personal home pages.

The directory `~username/public_html` on the Unix file system maps to the URL `http://hostname/~username/` via the web. So if your login name is stu`03` and you are using the PerlClass.com training server at `perlclass.fini.net`, you can access your web pages at `http://perlclass.fini.net/~sty03`. Of course, you will need to replace both the hostname and username to match your specific set-up.

23.2 The CGI directory

CGI scripts are usually kept in a separate directory from plain HTML files. This directory is most commonly called `cgi-bin` (the "bin" stands for "binary" but really just means "executable files", whether compiled binaries or interpreted scripts such as Perl programs). The web server is usually set up so that you only have permission to run CGI programs from the `cgi-bin` directory, for security reasons.

1. Change to your `public_html` directory
2. If you type **ls** to get a directory listing, you will see that you have several HTML files here, as well as a `cgi-bin` directory.
3. Change to your `cgi-bin` directory and type **ls**, and you will see that the example scripts for this course are already installed here.

If you were setting this up for yourself, you would need to be sure of the following:

1. That your home directory is world executable
2. That your `public_html` directory is world executable
3. That all your HTML files are world readable
4. That your `cgi-bin` directory is world executable - note that it is not compulsory to have a `cgi-bin` directory - some server configurations allow you to execute a CGI script from any directory.
5. That all your CGI scripts are world readable and executable

23.3 The HTTP headers

Every CGI script must output an HTTP header giving a MIME content type, such as `Content-type: text/html`, with a blank line after it:

```
print "Content-type: text/html\n\n";
```

Put this at the top of every CGI script, as the first thing that's printed.

> **ADVANCED**
>
> If your output is of another MIME type, you should print out the appropriate `Content-type:` header - for instance, a CGI program which outputs a random GIF image would use `Content-type: image/gif`

23.4 HTML output

Any other output of your script will be sent back to the web browser just as you specify. Since we're outputting content of the type `text/html` we should make our scripts output HTML:

```
print "<h1>Hello, world!</h1>\n";
```

The above example is already in your `cgi-bin` directory as `hello.cgi`.

23.5 Running and debugging CGI programs

When writing CGI programs, there are many problems which may affect their execution. Since these will not always be easily understood by examining the web browser output, there are other ways to check how your program is running:

1. First, check that your program runs by running it from the command line. It may be that you've made a syntax error, or that your program has the wrong permissions
2. Second, try opening it in a browser. If your program runs on the command line but does not output content to the browser, you may have forgotten to print out the `Content-type: text/html` header, or forgotten to leave a blank line between the header and the body, or may have made an error in your HTML output.
3. Thirdly, check the web server's log files. Where these are will vary from system to system. On our system, they're in `/var/log/apache`, and you can check them using **cat**, **less**, **tail**, or any other tool of your choice. If you don't know what these commands do, check their manual pages by typing **man cat**, **man less**, etc.

23.5.1 Exercises

1. Look at the output of the `hello.cgi` script by pointing a web browser (such as Netscape) at `http://`*`hostname`*`/~train`*`XX`*`/cgi-bin/hello.cgi` (replace *`hostname`* with the hostname or IP address of the training server, and *`XX`* with your number)
2. Modify `hello.cgi` to set a variable `$name` and include that name in the greeting. (Don't forget to `use strict;`)
3. Run your modified `hello.cgi` from the command line to ensure that it runs.
4. Press the `Reload` button in your browser to see if your modifications worked correctly.

23.6 Quoting made easy

It can be annoying to output HTML using double quotes. You may find yourself doing things like this:

```
print "<img src=\"$img\" alt=\"$alttext\">\n";
print "<a href=\"$url\">A hypertext link</a>\n";
```

Escaping all those quotes with backslashes can get tedious and unreadable. Luckily, there are a couple of ways around it.

23.6.1 Here documents

“Here”" documents allow you to print everything until a certain marker is found:

```
print <<"END";
<img src="$img" alt="$alttext">
<a href="$url">A hypertext link</a>
END
```

You can specify what end marker you want on in the `print` statement.

The fact that the marker is in double quotes means that the material up until the end marker is found will undergo interpolation in the same way as any double-quoted string. If you use single quotes, it'll act like a single-quoted string, and no interpolation will occur.

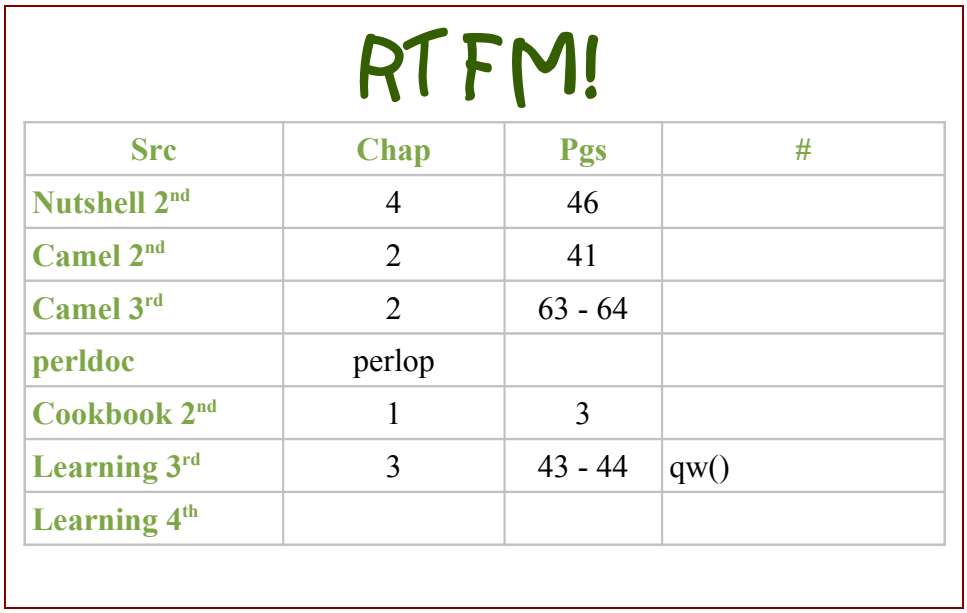
ADVANCED

If you use backticks, it will execute each line via the shell.

The end marker must be on a line by itself, at the very start of the line. Note also that the `print` statement has a semi-colon on the end.

23.7 Pick your own quotes

Another way of avoiding excessive backslashes in your code is to use the `qq()` or `q()` operators/functions.



Src	Chap	Pgs	#
Nutshell 2nd	4	46	
Camel 2nd	2	41	
Camel 3rd	2	63 - 64	
perldoc	perlop		
Cookbook 2nd	1	3	
Learning 3rd	3	43 - 44	qw()
Learning 4th			

```
print qq(<img src="$img" alt="$alttext">\n);
print qq(<a href="$url">A hypertext link</a>\n);
```

Like the matching and substitution operators `m//` and `s///`, the quoting operators can use just about any character as a delimiter:

```
print qq(<a href="$url">A hypertext link</a>\n);
print qq!<a href="$url">A hypertext link</a>\n!;
print qq[<a href="$url">A hypertext link</a>\n];
print qq#<a href="$url">A hypertext link</a>\n#;
```

If the opening delimiter is a bracket type character, the closing delimiter will be the matching bracket.

Always choose a delimiter that isn't likely to be found in your quoted text. A slash, while common in non-HTML uses of the function, is not very useful for quoting anything containing HTML closing tags like `</p>`.

23.8 Exercises

The following exercises practice using CGI to output different Perl data types (as taught in Introduction to Perl) such as arrays and hashes. You may use plain double quotes, ``here'' documents, or the quoting operators as you see fit.

1. Write a CGI program which creates an array then outputs the items in an unordered list (HTML's `<ul>` element) using a foreach loop. If you need help with HTML, there's a cheat sheet in Appendix D.
2. Modify your program to print out the keys and values in a hash, like this:
   - Name is Fred
   - Sex is male
   - Favorite colour is blue
3. Change your CGI program so that you output a table instead of an unordered list, with the keys in one column and the values in another. An example of how this could be done is in `cgi-bin/hashtable.cgi`

23.9 Environment variables

In Perl, there is a special variable called `%ENV` which contains all the environment variables which are set.

When a web server runs a CGI program, certain environment variables are set to provide information about the web server, the request made by the user agent, and other pertinent information.

Examples of environment variables available to your CGI script include HTTP_USER_AGENT which describes the user agent or browser used to make the request, and HTTP_REFERER, which indicates the referring page (if any).

23.9.1 Exercises

1. Modify your table-printing script from the previous exercise to print out the hash `%ENV`.
2. The `HTTP_USER_AGENT` environment variable contains the type of browser used to request the CGI script.
   - Write a script which prints out the user agent string for the requesting browser
   - Take a look at what various browsers report themselves as -- try Netscape, Internet Explorer, or Lynx from the Unix command line. You will note that Microsoft browsers purport to be "Mozilla compatible" (i.e. compatible with Netscape).
   - Use a regular expression to determine when a certain browser (for instance, Microsoft Internet Explorer) is being used, and output a message to the user.
3. The `HTTP_REFERER` (yes, it's spelt incorrectly in the protocol definition) environment variable contains the URL of the page from which the user followed a link to your CGI program. If you call up your CGI program by typing its URL straight into the browser, the `HTTP_REFERER` will be an empty string. Create an HTML page that points to your CGI program and see what the `REFERER` environment variable says.

23.10 Chapter summary

- CGI scripts are programs written in Perl or any other language that output web content such as HTML pages
- CGI scripts must output a Content-type header and a blank line before anything else
- Debugging techniques for CGI:
  - Run the script from the command line
  - Try opening it in the browser
  - Check the logs
- Various techniques are available for quoting text, including "here" documents and Perl quoting functions such as `qq()`.
- The `%ENV` special variable can be used to access environment variables via CGI scripts, including such variables as HTTP_USER_AGENT and HTTP_REFERER

Chapter 24: Accepting and processing form input

In this chapter...

CGI programs are often used to accept and process data from HTML forms. In this section, we take a quick look at HTML forms and use the `CGI` module to parse form data.

24.1 A quick look at HTML forms

To be able to use CGI to accept user input, you will probably need to understand HTML forms. There's an HTML cheat-sheet in Appendix D of these notes, but here's a brief run-down of the major parts of HTML forms:

24.2 The FORM element

The FORM element is a block level element - that means that the browser will present it on a new line, like it does with headings and paragraphs.

The FORM element's attributes include:

Table 4-1. FORM element attributes

Attribute	Example	Description
METHOD	METHOD="POST"	The HTTP method to use to send the form's contents back to the web server. It can be POST or GET -- the differences are explained the the HTTP cheat sheet appendix.
ACTION	ACTION="../cgi-bin/myscript.cgi"	The relative or absolute URL of the CGI program which is to process the form's data

Other attributes exist, but will not be used in this course.

24.3 Input fields

Some of the input fields you can use in your form include:

24.3.1 TEXT

A text input field `<INPUT TYPE="TEXT" NAME="email_address">`

24.3.2 CHECKBOX

Creates a yes/no checkbox. Saying `CHECKED` will make it checked by default.

```
<INPUT TYPE="CHECKBOX" NAME="send_email" CHECKED>
```

24.3.3 SELECT

Creates a drop-down list of items. Saying `SELECT MULTIPLE` will allow for multiple choices to be made.

```
<SELECT NAME="hobbies">
    <OPTION VALUE="philately">Philately</OPTION>
    <OPTION VALUE="gardening">Gardening</OPTION>
    <OPTION VALUE="programming">Programming</OPTION>
    <OPTION VALUE="cookery">Cookery</OPTION>
    <OPTION VALUE="reading">Reading</OPTION>
    <OPTION VALUE="bushwalking">Bushwalking</OPTION>
</SELECT>
```

24.3.4 SUBMIT

Creates a button which, when pressed, will submit the form.

```
<INPUT TYPE="SUBMIT" VALUE="Press me!">
```

24.4 The CGI module

24.4.1 What is a module?

A module is a collection of useful functions which you can use in your programs. They are written by Perl people worldwide, and distributed mostly through CPAN, the Comprehensive Perl Archive Network.

Perl modules save you heaps of time - by using a module, you save yourself from "reinventing the wheel". Perl modules also tend to save you from making silly mistakes again and again while you try to figure out how to do a given task.

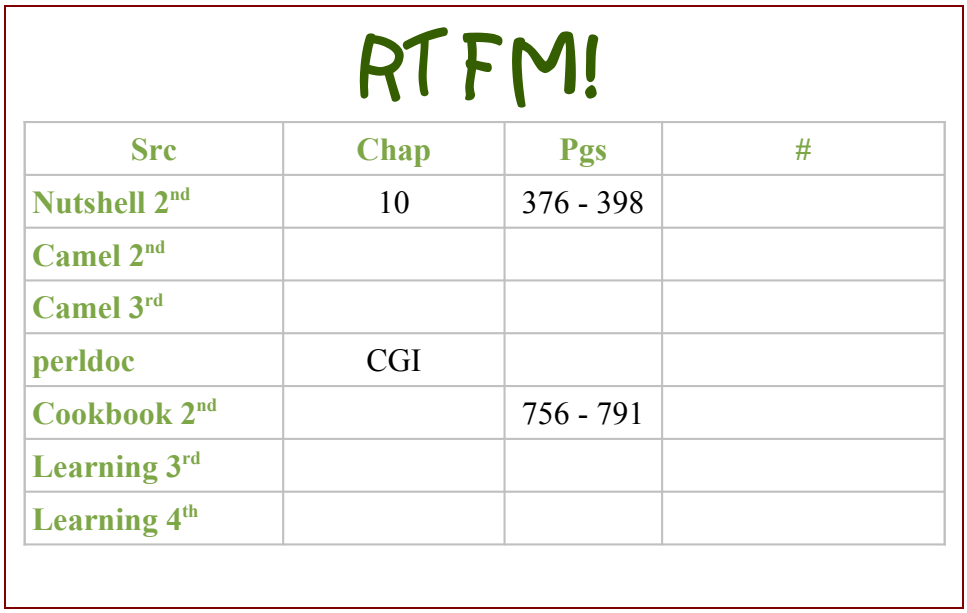
One common (but fiddly) task in CGI programming is taking the parameters given in an HTML form and turning them into variables that you can use.

The parameters from an HTML form are encoded in this "percent-encoded" format:

```
name=Kirrily&company=Netizen%20Pty.%20Ltd.
```

If you use the POST method, these parameters are passed via STDIN to the CGI script, whereas GET passes them via the environment variable `QUERY_STRING`. This means that as well as simply parsing the character string, you need to know where to look for it as well.

The easiest way to parse this parameter line is to use `CGI` module.



Src	Chap	Pgs	#
Nutshell 2nd	10	376 - 398	
Camel 2nd			
Camel 3rd			
perldoc	CGI		
Cookbook 2nd		756 - 791	
Learning 3rd			
Learning 4th			

24.4.2 Using the `CGI` module

To use the `CGI` module, simply put the statement `use CGI;` at the top of your script, thus:

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

use strict;
use CGI;
```

24.4.3 Accepting parameters with `CGI`

To accept form parameters into our CGI script as variables, we can say that we specifically want to use the `params` part of the CGI module:

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

use strict;
use CGI 'param';
```

This provides us with a new subroutine, `param`, which we can use to extract the value of the HTML form's fields.

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

use strict;
use CGI 'param';

my $name = param('name');
print "Content-type: text/html\n\n";
print "Hello, $name!";
```

24.4.4 Debugging with the `CGI` module's offline mode

When you run a CGI script from the command line, you will see a prompt like this:

```
(offline mode: enter name=value pairs on standard input)
```

This allows you to enter parameters in the form `name=value` for testing and debugging purposes. Use **CTRL-D** (the Unix end-of-file character) to indicate that you are finished.

```
(offline mode: enter name=value pairs on standard input)
name=fred
age=40
^D
```

24.4.5 Exercises

1. Write a simple form to ask the user for their name. Type in the above script and see if it works.
2. Add some fields to your form, including a checkbox and a drop down menu, and print out their values. What are the default true/false values for a checkbox?
3. What happens if you use the `SELECT MULTIPLE` form functionality? Try assigning that field's parameters from it to an array instead of a scalar, and you will see that the data is handled smoothly by the `CGI` module. Print them out using a `foreach` loop, as in earlier exercises.

24.5 Practical Exercise: Data validation

Your trainer will now demonstrate and discuss the use of `CGI` for validation of data entered into a web form. An example form is in your `public_html` directory as `validate.html` and the validation CGI script is available in your `cgi-bin` directory as `validate.cgi`.

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

use strict;
use CGI 'param';

print "Content-type: text/html\n\n";

my @errors;

push (@errors, "Year must be numeric") if param('year') =~ /\D/;
push (@errors, "You must fill in your name") if param('name') eq "";
push (@errors, "URL must begin with http://")
           if param('url') !~ m!^http://!;

if (@errors) {
        print "<h2>Errors</h2>\n";
        print "<ul>\n";
        foreach (@errors) {
                print "<li>$_\n";
        }
        print "</ul>\n";
} else {
        print "<p>Congratulations, no errors!</p>\n";
}
```

24.5.1 Exercises

1. Open the form for the validation program in your browser. Try submitting the form with various inputs.

24.6 Practical Exercise: Multi-form "Wizard" interface

Your trainer will now demonstrate and discuss how you can use what you've just learned to create a multi-form "wizard" interface, where values are remembered from one form to the next and passed using hidden fields.

```
<INPUT TYPE="HIDDEN" VALUE="..." NAME="...">
```

Source code for this example is available as `cgi-bin/wizard.cgi`.

First, we print some headers and pick up the "step" parameter to see what step of the wizard interface we're up to. We have four subroutines, named `step1` through `step4`, which do the actual work for each step.

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

use strict;
use CGI 'param';

print <<"END";
Content-type: text/html

<html>
<body>
<h1>Wizard interface</h1>
END

my $step = param('step') || 0;

step1() unless $step;
step2() if $step == 2;
step3() if $step == 3;
step4() if $step == 4;

print <<"END";
</body>
</html>
END
```

Here are the subroutines. The first one is fairly straightforward, just printing out a form:

```
#
# Step 1 -- Name
#

sub step1 {
        print qq(
                <h2>Step 1: Name</h2>
                <p>
                What is your name?
                </p>
                <form method="POST" action="wizard.cgi">
                <input type="hidden" name="step" value="2">
                <input type="text" name="name">
                <input type="submit">
                </form>
        );
}
```

Steps 2 through 4 require us to pick up the CGI parameters for each field that's been filled in so far, and print them out again as hidden fields:

```
#
# Step 2 -- Quest
#

sub step2 {
        my $name = param('name');
        print qq(
                <h2>Step 2: Quest</h2>
                <p>
                What is your quest?
                </p>
                <form method="POST" action="wizard.cgi">
                <input type="hidden" name="step" value="3">
                <input type="hidden" name="name" value="$name">
                <input type="text" name="quest">
```

```
                <input type="submit">
                </form>
        );
}


#
# Step 3 -- favorite colour
#

sub step3 {
        my $name = param('name');
        my $quest = param('quest');

        print qq(
                <h2>Step 3: Silly Question</h2>
                <p>
                What is the airspeed velocity of an unladen swallow?
                </p>
                <form method="POST" action="wizard.cgi">
                <input type="hidden" name="step" value="4">
                <input type="hidden" name="name" value="$name">
                <input type="hidden" name="quest" value="$quest">
                <input type="text" name="swallow">
                <input type="submit">
                </form>
        );
}
```

Step 4 simply prints out the values that the user entered in the previous steps:

```
#
# Step 4 -- finish up
#

sub step4 {
        my $name = param('name');
        my $quest = param('quest');
        my $swallow = param('swallow');
```

```
        print qq(
                <h2>Step 4: Done!</h2>
                <p>
                Thank you!
                </p>
                <p>
                Your name is $name.  Your quest is $quest.  The
airspeed
                velocity of an unladen swallow is $swallow.
                </p>
        );
}
```

24.6.1 Exercises

1. Add another question to the `wizard.cgi` script.

24.7 Practical Exercise: File upload

CGI can also be used to allow users to upload files. Your trainer will demonstrate and discuss this. Source code for this example is available in your cgi-bin directory as upload.cgi

First off, you need to specify an encoding type in your form element. The attribute to set is ENCTYPE="multipart/form-data".

```
<html>
<head>
<title>Upload a file</title>
</head>
<body>
<h1>Upload a file</h1>

Please choose a file to upload:

<form action="upload.cgi" method="POST" enctype="multipart/form-
data">
<input type="file" name="filename">
<input type="submit" value="OK">
</form>
</body>
</html>
```

CGI handles file uploads quite easily. Just use param() as usual. The value returned is special -- in a scalar context, it gives you the filename of the file uploaded, but you can also use it in a filehandle.

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

use strict;
use CGI 'param';

my $filename = param('filename');
my $outfile = "outputfile";

print "Content-type: text/html\n\n";
```

```
# There will probably be permission problems with this open
# statement unless you're running under cgiwrap, or your script
# is setuid, or $outfile is world writable.  But let's not worry
# about that for now.

open (OUTFILE, ">$outfile") || die "Can't open output file: $!";

# This bit is taken straight from the CGI.pm documentation --
# you could also just use "while (<$filename>)" if you wanted

my ($buffer, $bytesread);
while ($bytesread=read($filename,$buffer,1024)) {
        print OUTFILE $buffer;
}

close OUTFILE || die "Can't close OUTFILE: $!";

print "<p>Uploaded file and saved as $outfile</p>\n";

print "</body></html>";
```

24.8 Chapter summary

- The `CGI` module can be used to parse data from HTML forms
- Its most common use is parameter parsing; other functions are also available
- To use it, type `use CGI 'param';` at the top of your script
- Obtain each item of data using the `param()` function
- `CGI` can be used to implement web applications of any complexity, including data validation, multi-form wizards, file upload, and more

Chapter 25: Security issues

In this chapter...

In this section we examine some security issues arising from the use of CGI scripts, including authentication and access control, and the risk of tainted data and how to avoid it.

25.1 Authentication and access control for CGI scripts

A common question asked by new CGI programmers is "How do I protect my web site with a CGI script?" There are various ways to use CGI programs to ask for usernames and passwords and perform authentication, but in fact the best way to perform authentication and access control comes with your web server and doesn't require any programming at all.

The reason that password protection is often connected with CGI programs is that CGI programs are more likely to interact with the web server's underlying file system, backend databases, or other things which need to be kept secure. Many programmers assume that because CGI can be used for password protection, it is the right choice for the job. This is not necessarily true.

One of the best ways to password protect web pages is by using the web server's own authentication and access control mechanisms. Since we're using the Apache web server, we'll look at how to do it with that.

25.1.1 Why is CGI authentication a bad idea?

Authentication (i.e. username and password checking) is hard to do correctly in CGI. Some common pitfalls include:

- Username and password strings are sent as parameters in a GET query, and end up in the URL (eg `http://example.com/my.cgi?username=fred&password=secret`). These details can then end up in peoples' bookmark files, other sites' referer logs, and so on.
- Sometimes username and password details are passed back and forth using "cookies". Many users choose to have cookies disabled due to privacy concerns, and the website will therefore be unusable to them. No such problem exists with HTTP authentication via the web server

On the other hand, the main disadvantage of HTTP authentication is that the authentication tokens remain active until the user shuts their browser down. This can be a problem in public computer labs and other locations where users may share PCs.

25.2 HTTP authentication

If a web page or CGI script requires a username and password to view it, the HTTP conversation between the client and the server goes like this:

1. The user specifies a URL
2. The user agent connects to port 80 of the HTTP server
3. The user agent sends a request such as `GET /index.html`
4. The user agent may also send other headers
5. The HTTP server realises that authentication must be performed {usually by looking up configuration files}
6. The HTTP server returns a status code 401, meaning "Unauthorized", and also a header saying `WWW-Authenticate:` and the name of the authentication domain, for instance "Acme Widget Co. Staff". This usually appears in the browser's dialog box as "Please provide a username and password for Acme Widget Co. Staff".
7. The browser presents a dialog box or other means by which the user can enter their username and password, which the user fills in then clicks "OK"
8. The browser sends a new request, this time including an extra header saying `Authorization:` and the appropriate credentials
9. If the HTTP server finds that the credentials are valid, it sends back the resource requested and closes the connection
10. Otherwise, it sends back another response with status code `401` (and probably a body containing an error message), which the user agent should recognise as meaning that the authentication failed, and display the body.

25.3 Access control

The way access control is handled varies from one web server to another. If your web server is not Apache, you will need to contact your web server administrator or read the documentation it came with, as only Apache is covered in this course.

Apache implements HTTP authentication with the use of a password file and either server configurations or a `.htaccess` file in the web directory, which contains server configuration directives. Our server has been set up to allow you to use the `.htaccess` file.

A password file has already been set up for your use. It's `/etc/apache/training.passwd` and uses the same usernames and passwords as your login accounts. You can look at it by typing `cat /etc/apache/training.passwd`

To use this password file, create a file in your `public_html` directory called `.htaccess`, containing the following text:

```
AuthType Basic
AuthName "Secret stuff"
AuthUserFile /etc/apache/training.passwd
require valid-user
```

This authentication will apply to the directory in which the `.htaccess` file is placed and any subdirectories.

25.3.1 Exercises

1. Create a `.htaccess` file in your `public_html` directory, as above
2. Use your web browser to request one of your HTML files or CGI scripts, and observe the authentication process
3. Why would it be a bad idea to put the password file in the same directory as the web pages or CGI scripts?

25.4 Tainted data

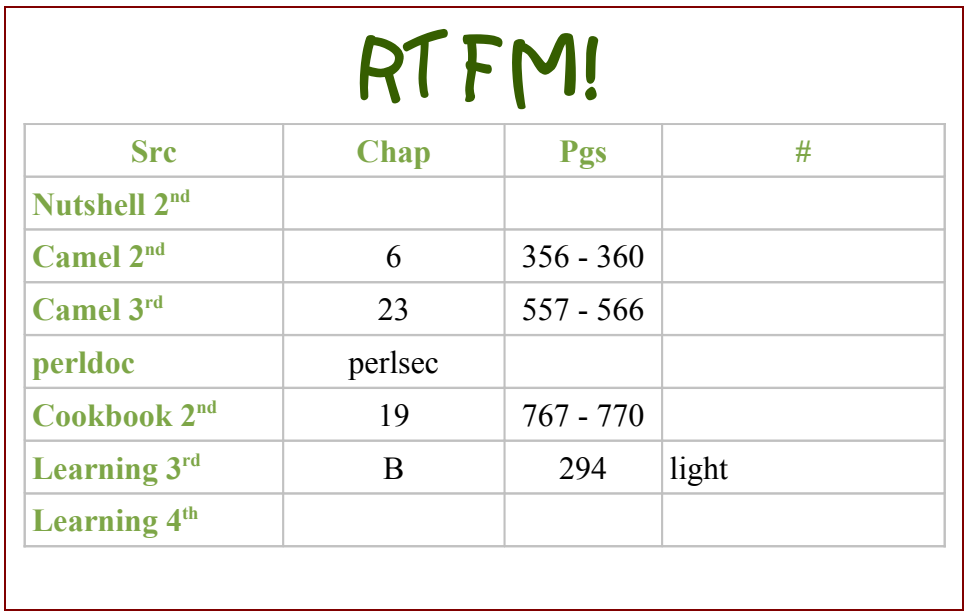
Sometimes you will want to write a CGI script which interacts with the system. This can result in major security risks if the commands executed on the system are based on user input. Consider the example of a finger program which asked the user who they wanted to finger.

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

use strict;

print "Who do you want to finger? ";
my $username = <STDIN>;
print `finger $username`;   # backticks execute shell command
```

Imagine if the user's input had been `skud; cat /etc/passwd`, or worse yet, `skud; rm -rf /` The system would perform both commands as though they had been entered into the shell one after the other.

Luckily, Perl's `-T` flag can be used to check for unsafe user inputs.

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



Src	Chap	Pgs	#
Nutshell 2nd			
Camel 2nd	6	356 - 360	
Camel 3rd	23	557 - 566	
perldoc	perlsec		
Cookbook 2nd	19	767 - 770	
Learning 3rd	B	294	light
Learning 4th			

`-T` stands for "taint checking". Data input by the user, either via the command

line or an HTML form, 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.

The only thing that will clear tainting is referencing substrings from a regexp match. **perldoc perlsec** contains a simple example of how to do this, about 7 pages down. Read it now, and use it to complete the following exercises.

Note that you'll also have to explicitly set `$ENV{'PATH'}` to something safe (like `/bin`) as well.

25.4.1 Exercises

1. The HTML file `finger.html` asks the user for an account name about which to obtain information {using the Unix system's `finger` command}. It calls the CGI script `cgi-bin/finger.cgi` which uses taint checking.
2. Why is the data input by the user tainted?
3. Add a `-T` flag to the shebang line of `finger.cgi` so that the script performs taint checking
4. Try re-submitting the form - it should fail
5. To untaint the data, you need to clean up any unwanted characters. Use some code similar to that in **perldoc perlsec** to remove anything other than alphanumeric characters and assign the valid part of the user input to a new variable.

25.5 cgiwrap

Many large sites, such as ISPs and educational institutions, require users to run their CGI scripts using a program called **cgiwrap**. This program causes the CGI script to execute as if being run by the owner, instead of the web server's user ID. What this means is that the script will have permission to read and write the user's files, and will not be able to cause any damage on the system that the user could not cause.

25.6 Secure HTTP

Another somewhat related topic is secure HTTP, which uses the HTTPS protocol to open a secure connection and encrypts all data between the web client and server. This is often used to make online credit card transactions more secure.

CGI scripts can be run on a secure server exactly as they would run on any other server.

25.7 Chapter summary

- HTTP authentication can be used to password protect web pages
- The Apache web server implements HTTP authentication. This can be configured via a `.htaccess` file
- There is a security risk from tainted data --- data entered by a user which is used for subsequent system interaction
- Perl has built-in checking for tainted data, which can be turned on my using the `-T` command line switch
- Data can be untainted by referencing a substring in a match, as shown in **perldoc perlsec**.
- Some web servers use **cgiwrap** to run CGI scripts under their owner's user ID.
- Secure HTTP can be used to provide an encrypted channel of communication between the web client and server.

Chapter 26: Other related Perl modules

In this chapter...

In this section we are briefly introduced to Perl modules which may be useful to us in developing CGI applications, including modules for failing gracefully, encoding and decoding URLS, and filling in templates.

26.1 Useful Perl modules

There are several common problems faced by CGI programmers: failing gracefully, creating valid URLs from any text, using a template to insert variables into HTML, sending email based on CGI parameters, et cetera. Since these problems are so common, people have written modules to solve them. This section explains some of the most useful modules to save you from having to re-invent the wheel.

Each of these modules can be downloaded from CPAN (the Comprehensive Perl Archive Network) (http://www.perl.com/CPAN) and installed either using the CPAN module distributed with Perl, or by following the steps described in the `README` file distributed with each module.

26.2 Failing gracefully with `CGI::Carp`

The errors given in the web server's error logs are not always easy to read and understand. To make life easier, we can use a Perl module called `CGI::Carp` to add timestamps and other handy information to the logs.

```
use CGI::Carp;
```

We can also make our errors go to a separate log, by using the `carpout` part of the module. This needs to be done inside a `BEGIN` block in order to catch compiler errors as well as ones which occur at the interpretation stage.

```
BEGIN {
        use CGI::Carp qw(carpout);
        open(LOG, ">>cgi-logs/mycgi-log") ||
                die("Unable to open mycgi-log: $!\n");
        carpout(LOG);
}
```

Lastly, we can cause any fatal errors to have their error messages and diagnostic information output directly to the browser:

```
use CGI::Carp 'fatalsToBrowser';
```

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd	8	192	
Camel 2nd	7	385	
Camel 3rd	32	878	
perldoc	Carp		
Cookbook 2nd	12	473 - 475	
Learning 3rd			
Learning 4th			

26.2.1 Exercise

1. Use the `CGI::Carp` module in one of your scripts
2. Deliberately cause a syntax error, for instance by removing a semi-colon or quote mark, or inserting a `die ("Argh!");` statement, and see what happens

26.3 Encoding URIs with `URI::Escape`

Sometimes we want to output anchor tags `<A HREF="...">` referring to another CGI script, and pass parameters along with it, thus:

```
<A HREF="lookup.cgi?title=Programming Perl&publisher=O'Reilly">
O'Reilly's Programming Perl
</A>
```

However, spaces and apostrophes aren't allowed in URIs, so we have to encode them into the "percent-encoded" format. This format replaces most non-alphanumeric characters with two hexadecimal digits. For instance, a space becomes `%20` and a tilde becomes `%7E`.

The Perl module we use to encode URIs in this manner is `URI::Escape`. Its documentation is available by typing **perldoc URI::Escape**.

Use it as follows:

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

use strict;
use URI::Escape;

my $book_lookup =
"lookup.cgi?title=Programming Perl&publisher=O'Reilly";

my $encoded_url = uri_escape($address);
my $original_url = uri_unescape($encoded_url);
```

26.3.1 Exercise

1. Try out the above script `cgi-bin/escape.cgi` you'll need to print out the values of `$encoded_url` and `$original_url`

26.4 Creating templates with `Text::Template`

By this stage in the day you have probably spent a great deal of time outputting HTML either via a long list of `print` statements or by using a "here document" or other shortcut. What if you wanted to have a template HTML output file which was filled in with the appropriate variables?

Luckily, there is a Perl module to do this, called `Text::Template`. Unluckily, it uses a concept we haven't covered yet, but which we will now explain.

`Text::Template` is different to the other modules we have used so far today, in that it is an *object oriented* module. Object oriented Perl modules can be very powerful, but require some background knowledge to understand how they work.

26.4.1 Introduction to object oriented modules

Before embarking on this task, we need to have an understanding of how Perl's object-oriented modules work. Not all modules are object oriented (`URI::Escape`, for example, is not), and some can be used either way (CGI is one of these), but some require us to work with them in this way.

A software object, like a real-life object, has attributes (things that describe the object) and methods (things you can do with, or to, the object). Consider the real-life example of a cup:

Table 6-1. Attributes and Methods of a cup

Object	Attributes	Methods
Cup	• colour • handle (does it have one?) • contents (water, coffee, etc) • fullness	• drink from it • fill it up • smash it

Note that when you smash a cup, you aren't smashing the generic class of cups, but rather a specific instance - *this* cup, not "cups in general". This is what we call an *instance of a class* -- remember that, as we'll use it later.

26.4.2 Using the `Text::Template` module

Like the cup, our text template has attributes and methods.

Table 6-2. Attributes and Methods of `Text::Template`

`Text::Template`	• `TYPE` - the type of template it is, eg a file, a string you created earlier, etc • `SOURCE` - the filehandle or variable name in which the template can be found	• `fill_in()` - fill in the template

Before we can actually use these attributes and methods in any useful way, we have to create a new instance of the class. This is the same as how we needed a specific cup, rather than the general class of cups.

```
# using the class in general
use Text::Template;

# instantiating the class and setting some attributes for the new
instance
my $letter = new Text::Template{'TYPE' => 'FILE', 'SOURCE' =>
'letter.tmpl'};
```

We can then perform a method on it, thus:

```
my $finished_letter = $letter->fill_in();
```

This will fill in any variables found in the template file.

26.4.3 Exercise

1. Type **perldoc Text::Template** and look at the documentation for this module
2. `cgi-bin/letter.cgi` implements the example above. Examine the source code.
3. Make some changes to the letter template and see if they work.

26.5 Sending email with `Mail::Mailer`

The `Mail::Mailer` module can be used to send email from a CGI script (or, for that matter, any script). Like `Text::Template`, it is an Object Oriented module. The object it creates is a "mailer" object, which can be opened and then printed to as if it were a filehandle.

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

use strict;
use Mail::Mailer;

my $mailer = new Mail::Mailer;

# the open() method takes a hash reference with keys which are mail
# header names and values which are the values of those mail headers

$mailer->open( {
        From    =>      'fred@example.com',
        To      =>      'barney@example.com',
        Subject =>      'Web form submission'
} );

# we can print to $mailer just as we would print to STDOUT or any
# other file handle...

print $mailer qq(
Dear Barney,

Here is a form submission from your website:

Name:           $name
Email:          $email
Comments:       $comments

Love, Fred.
);

$mailer->close();
```

ADVANCED

You can also open a pipe to **sendmail** directly, but doing this correctly can be difficult. This is why we recommend `Mail::Mailer` to avoid re-inventing the wheel.

26.5.1 Exercises

1. Create an HTML form with fields for name, email and comment
2. Use the above script (`cgi-bin/mail.cgi`) to mail the results of the script to yourself. You will need to edit it to work fully:
   - Use CGI.pm to pick up the parameters
   - Change the email address to your own address
   - Print out a "thank you" page once the form has been submitted -- don't forget the Content-type header

26.6 Chapter Summary

- The `CGI::Carp` module can be used to help CGI programs fail gracefully
- The `URI::Escape` module can be used to encode and decode percent-encoded URLs
- The `Text::Template` module can be used to easily fill in text templates, including HTML templates.
- The `Mail::Mailer` module can be used to send email based on the information entered in an HTML form
- All these modules can be downloaded from CPAN, the Comprehensive Perl Archive Network

Chapter 27: Conclusion

In the conclusion...

Summing up and various paths for further study.

27.1 Day 1: What you've learned

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

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

27.2 Day 2: What you've learned

Now you've completed PerlClass.com's *Intermediate Perl* module, you should be confident in your knowledge of the following fields:

- File I/O, including opening files and directories, opening pipes, finding information about files, recursing down directories, file locking, and handling binary data
- How to use advanced regular expression techniques such as multiline matching and backreferences
- The use of various Perl functions
- System interaction, including: system calls, the backtick operator, interacting with the file system, dealing with users and groups, dealing with processes, network communications, and security considerations
- Advanced Perl data structures and references

27.3 Day 3: What you've learned

Now you've completed PerlClass.com's CGI Programming in Perl module, you should be confident in your knowledge of the following fields:

- What CGI is
- How HTTP allows web user agents (browsers) to communicate with web servers and retreive documents
- How to perform HTTP requests by using **telnet** to connect to the web server
- How to generate simple web pages using Perl
- How to access environment variables from CGI scripts
- Various methods of quoting text, including "here" documents and the `qq()` type functions
- How to process data from HTML forms using the CGI module
- How to use the CGI module for applications such as data validation, simple "wizard" interfaces, and file uploads
- Security issues related to CGI programming, including authentication and access control, dealing with tainted data, secure web servers, etc.
- The use of various Perl modules related to CGI programming, including CGI::Carp, URI::Escape, Text::Template, and Mail::Mailer
- A basic understanding of object oriented Perl modules

27.4 Day 4: What you've learned

Now you've completed PerlClass.com's Database Programming with Perl module, you should be confident in your knowledge of the following fields:

- Database terminology, including tables and relationships, fields and records, etc
- Flat file database manipulation including delimited and CSV text files
- Basic SQL queries, including `SELECT`, `INSERT`, `DELETE`, and `UPDATE` queries
- Features of MySQL, where to get MySQL from, and how to set up MySQL databases
- Using the MySQL command line client to perform SQL queries
- Using Perl's DBI module to interact with databases
- Applying Perl skills from previous training modules to create database applications

27.5 Where to now?

To further extend your knowledge of Perl, you may like to:

- Borrow or purchase the books listed 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
- Install Perl and MySQL (or other database servers) on your home or work computer
- Install Perl and a web server such as Apache on your home or work computer
- Practice using Perl for CGI programming on a daily basis
- Practice using Perl to interact with databases
- Join a Perl user group such as Perl Mongers (http://www.pm.org/)
  - Richmond Perl Mongers (http://richmond.pm.org/)
  - Hampton Roads Perl Mongers (http://norfolk.pm.org/)

27.6 Further reading

27.6.1 Books

- Alligator Descartes & Tim Bunce, "Programming the Perl DBI", O'Reilly and Associates, 2000
- Randy Jay Yarger, George Reese & Tim King, "mSQL and MySQL", O'Reilly and Associates, 1999
- 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.

27.6.2 Online

- The Perl homepage (http://www.perl.com/)
- The Perl Journal (http://www.tpj.com/)
- Perlmonth (http://www.perlmonth.com/) (online journal)
- Perl Mongers Perl user groups (http://www.pm.org/)
- comp.lang.perl.announce newsgroup
- comp.lang.perl.moderated newsgroup
- comp.lang.perl.misc newsgroup

Chapter 28: Unix cheat sheet

28.1 Some UNIX commands

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

Table A-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***
dddd	

ddd

Chapter 29: Editor cheat sheet

This summary is laid out as follows:

Table B-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.

29.1 vi

29.1.1 Running

```
% vi filename
```

29.1.2 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.

29.1.3 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.

29.1.4 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).

29.1.5 Help

`:help` **ENTER** might work. If not, then see the manpage.

29.2 pico

29.2.1 B.Running

```
% pico -w filename
```

29.2.2 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.

29.2.3 Exiting

Follow the menu bar, if you are in the midst of a command. Use **CTRL-X** from the main menu.

29.2.4 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. \\ \hline

29.2.5 Help

CTRL-G from the main menu, or just read the menu bar.

29.3 joe

29.3.1 Running

```
% joe filename
```

29.3.2 Using

- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-K** then **S** to save.

29.3.3 Exiting

- **CTRL-C** to exit without save.
- **CTRL-K** then **X** to save and exit.

29.3.4 Gotchas

Nothing in particular.

29.3.5 Help

CTRL-K then **H**.

29.4 jed

29.4.1 Running

```
% jed
```

29.4.2 Using

- Defaults to the emacs emulation mode.
- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-X** then **S** to save.

29.4.3 Exiting

CTRL-X then **CTRL-C** to exit.

29.4.4 Gotchas

Nothing in particular.

29.4.5 Help

- Read the menu bar at the top.
- Press **ESC** then **?** then **H** from the main menu.

Chapter 30: ASCII Pronunciation Guide

It is widely recognized that speaking about computing topics requires some common set of terms for communications, so computerese or technobabble describe this dialect. But it is less widely recognized that a dialect is necessary for unambiguously communicating about individual characters.

Table C-1. ASCII Pronunciation Guide

Character	Pronunciation
!	bang, exlamation
*	star, asterisk
$	dollar
@	at
%	percent
&	ampersand
"	double quote
'	single quote, tick, or forward quote
( )	open/close bracket, parentheses

<	less than, left angle bracket
>	greater than, right angle bracket
-	dash, hyphen, n-dash
.	dot, period
,	comma
/	slash, forward slash
\	backslash
:	colon
;	semicolon
=	equals
?	question mark
^	caret (pron. "carrot")
_	underscore
[ ]	open/close square bracket
{ }	open/close curly brackets, brace, squigglies, or squiggly brackets
|	pipe, bar, or vertical bar
~	tilde (pron."til-duh"), wiggle
`	backtick, backquote (below ~)

Chapter 31: HTML Cheat Sheet

The following table outlines a few HTML elements which may be useful to you. For more detail or for information about elements which are not listed here, consult one of the references listed below.

Table D-1. Basic HTML elements

Type of information	Markup
Paragraph	`<P> ... </P>`
Heading level 1	`<H1>This is a level 1 heading</H1>`
Heading level 2	`<H2>This is a level 2 heading</H2>`
Heading level 3	`<H3>This is a level 3 heading</H3>`
Heading level 4	`<H4>This is a level 4 heading</H4>`

Unordered (bulleted) list	`<UL>`
	`<LI>List item 1`
	`<LI>List item 2`
	`<LI>List item 3`
	`<LI>List item 4`
	`</UL>`

Ordered (numbered) list	`<OL>`
	`<LI>List item 1`
	`<LI>List item 2`
	`<LI>List item 3`
	`<LI>List item 4`
	`</OL>`

Table	`<TABLE BORDER>`
	`<TR> <-- "table row" -- >`
	`<TH>Heading for column 1</TH>`
	`<TH>Heading for column 2</TH>`
	`<TH>Heading for column 3</TH>`
	`</TR>`

	<TR> <-- "table row" -- > <TD>Data for row 1, column 1</TD> <TD>Data for row 1, column 2</TD> <TD>Data for row 1, column 3</TD> </TR> <TR> <-- "table row" -- > <TD>Data for row 2, column 1</TD> <TD>Data for row 2, column 2</TD> <TD>Data for row 2, column 3</TD> </TR> </TABLE>
Horizontal rule	<HR>
Anchor tag (hypertext link)	<A HREF="http://example.com/">Descriptive text</A>

For more information...

- HTMLhelp.org (http://htmlhelp.org/)
- The World Wide Web Consortium (W3C) (http://w3.org/)