

PerlClass.com's Perl Training Materials

**Christopher Hicks
and
Kirrily Robert**

Perl Training Materials

by Christopher Hicks

Copyright ©

1999-2000, Netizen Pty Ltd

2000 by Kirrily Robert

2001-2009 by Christopher Hicks

License

This book is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software..

This book is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this book; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA or go to <http://www.gnu.org/> .

This book was based on material under the Open Publications License available at <http://www.content.org/openpub/> .

File and Version Info

Monday, 18 May 2009 03:23:38 AM 511 pages

`/Users/chicks/Documents/PerlClass/perlClass-0.70.odt`

Table of Contents

Chapter 1: Overview.....	19
1.1 Assumed knowledge.....	20
1.2 Rough outline.....	21
1.3 Other topics we can discuss.....	22
1.4 Platform and version details.....	23
1.5 The course notes.....	24
1.6 Other materials.....	26
Chapter 2: What is Perl.....	27
2.1 Perl's name.....	28
2.2 Typical uses of Perl.....	29
2.2.1 Text processing.....	29
2.2.2 System administration tasks.....	29
2.2.3 CGI and web programming.....	29
2.2.4 Database interaction.....	29
2.2.5 Other Internet programming.....	29
2.2.6 Less typical uses of Perl.....	29
2.3 What is Perl like?.....	30
2.4 The Perl Philosophy.....	32
2.4.1 There's more than one way to do it.....	32
2.4.2 A correct Perl program.....	32
2.4.3 Three virtues of a programmer.....	32
2.4.3.1 Laziness.....	32
2.4.3.2 Impatience.....	32
2.4.3.3 Hubris.....	32
2.4.4 Three more virtues.....	33

2.4.5 Share and enjoy!.....	33
2.5 Parts of Perl.....	34
2.5.1 The Perl interpreter.....	34
2.5.2 Manuals.....	34
2.5.3 Perl Modules.....	34
2.6 CPAN.....	35
2.7 Slashdot.....	36
2.8 Chapter summary.....	37
Chapter 3: Creating a a Perl program.....	39
3.1 Logging into your account from Windows.....	40
3.2 Using perldoc.....	42
3.3 Using the editor.....	53
3.4 Our first Perl program.....	54
3.5 Running a Perl program from the command line.....	55
3.6 The "shebang" line.....	56
3.7 Comments.....	57
3.8 Command line options.....	58
3.9 Chapter summary.....	59
Chapter 4: Perl variables.....	61
4.1 What is a variable?.....	62
4.2 Variable names.....	63
4.3 Variable scoping and the strict pragma.....	64
4.3.1 Arguments in favour of strictness.....	64
4.3.2 Arguments against strictness.....	64
4.4 Using the strict pragma.....	66
4.5 More useful pragmas.....	67
4.6 Scalars.....	69
4.7 Double and single quotes.....	71
4.8 Exercises.....	73
4.9 Answers.....	74
4.10 Arrays.....	75
4.10.1 A quick look at context.....	78
4.10.2 What's the difference between a list and an array?.....	78
4.11 Exercises.....	80
4.11.1 Advanced exercises.....	80
4.12 Answers.....	81
4.12.1 Advanced Answer.....	81
4.13 Hashes.....	83
4.13.1 Initializing a hash.....	83

4.13.2 Reading hash values.....	84
4.13.3 Adding new hash elements.....	84
4.13.4 Other things about hashes.....	84
4.13.5 What's the difference between a hash and an associative array?.....	85
4.14 Exercises.....	86
4.15 Answers.....	87
4.16 Special variables.....	88
4.16.1 The first special variable, \$_.....	88
4.16.2 @ARGV - a special array.....	89
4.16.3 %ENV - a special hash.....	89
4.17 Exercises.....	90
4.18 Answers.....	91
4.18.1 Scalar answer.....	91
4.18.2 Array Answer.....	91
4.18.3 Hash Answer.....	91
4.19 Chapter summary.....	92
Chapter 5: Operators and functions.....	93
5.1 What are operators and functions?.....	94
5.2 Arithmetic operators.....	95
5.3 String operators.....	96
5.3.1 Exercises.....	96
5.4 Answers.....	97
5.4.1 Exercise 1.....	97
5.4.2 Exercise 2.....	97
5.4.3 Source to operate.pl.....	97
5.5 File operators.....	98
5.6 Other operators.....	99
5.7 Functions.....	100
5.7.1 Types of arguments.....	100
5.7.2 Return values.....	101
5.8 More about context.....	102
5.9 String manipulation.....	103
5.9.1 Finding the length of a string.....	103
5.9.2 Case conversion.....	103
5.9.3 chop() and chomp().....	103
5.9.4 String substitutions with substr().....	104
5.10 Numeric functions.....	105
5.11 Type conversions.....	106
5.12 Manipulating lists and arrays.....	107
5.12.1 Stacks and queues.....	107

5.12.2 Sorting lists.....	108
5.12.3 Converting lists to strings, and vice versa.....	108
5.13 Hash processing.....	109
5.14 Reading and writing files.....	110
5.15 Time.....	111
5.16 Exercises.....	112
5.17 Answers.....	113
5.17.1 Exercise 1.....	113
5.17.2 Exercise 3.....	113
5.17.3 Exercise 4.....	113
5.17.4 Exercise 5.....	114
5.17.5 Exercise 6.....	114
5.18 Chapter summary.....	115
Chapter 6: Conditional constructs.....	117
6.1 What is a block?.....	118
6.2 Scope.....	119
6.3 What is a conditional statement?.....	120
6.4 What is truth?.....	121
6.5 Comparison operators.....	122
6.5.1 Existence and Defined-ness.....	123
6.5.2 Boolean logic operators.....	124
6.5.3 Using boolean logic operators for flow control.....	126
6.6 Types of conditional constructs.....	128
6.6.1 if statements.....	128
6.6.2 while loops.....	129
6.6.3 for and foreach.....	129
6.7 Exercises.....	131
6.7.1 Answer.....	131
6.8 Practical uses of while loops: taking input from STDIN.....	133
6.9 Named blocks.....	135
6.10 Breaking out of loops.....	136
6.11 Chapter summary.....	137
Chapter 7: Subroutines.....	139
7.1 Introducing subroutines.....	140
7.2 Calling a subroutine.....	141
7.3 Passing arguments to a subroutine.....	142
7.4 Returning values from a subroutine.....	143
7.5 Exercises.....	144
7.6 Answers.....	145

7.6.1 Exercise 1.....	145
7.6.2 Exercise 2.....	145
7.6.3 Exercise 3.....	145
7.7 Chapter summary.....	147
Chapter 8: Regular expressions.....	149
8.1 What are regular expressions?.....	150
8.2 Regular expression operators and functions.....	151
8.2.1 m/PATTERN/ - the match operator.....	151
8.2.2 s/PATTERN/REPLACEMENT/ - the substitution operator.....	151
8.3 Binding operators.....	153
8.4 Metacharacters.....	154
8.4.1 Some easy metacharacters.....	154
8.5 Quantifiers.....	156
8.6 Greediness.....	157
8.7 Exercises.....	158
8.8 Answers.....	159
8.8.1 Exercise 1.....	159
8.8.2 Exercise 2.....	159
8.8.3 Exercise 3.....	159
8.9 Character classes.....	160
8.9.1 Exercises as a group.....	160
8.10 Alternation.....	161
8.11 The concept of atoms.....	162
8.12 Exercises.....	163
8.13 Answers.....	164
8.13.1 Exercise 1.....	164
8.13.2 Exercise 2.....	164
8.13.3 Exercise 3.....	164
8.14 split() function.....	166
8.15 Exercises.....	167
8.16 Answers.....	168
8.16.1 Exercise 1.....	168
8.16.2 Exercise 2.....	168
8.17 Chapter summary.....	169
Chapter 9: Practical exercises.....	171
9.1 Exercises.....	172
Chapter 10: File I/O.....	173
10.1 Assumed knowledge.....	174

10.2 Angle brackets - the line input and globbing operators.....	175
10.2.1 Exercises.....	177
10.2.1.1 Advanced exercises.....	177
10.3 Answers.....	178
10.3.1 Exercise 1.....	178
10.3.2 Exercise 2.....	178
10.3.3 Exercise 3.....	178
10.3.4 Advanced Exercise 1.....	179
10.4 open() and friends - the gory details.....	180
10.4.1 Opening a file for reading, writing or appending.....	180
10.4.2 Exercises.....	182
10.5 Answers.....	183
10.5.1 Exercise 1.....	183
10.5.2 Exercise 2.....	183
10.5.3 Exercise 3.....	183
10.5.4 Exercise 4.....	184
10.5.5 Exercise 5.....	184
10.6 Reading directories.....	185
10.7 Exercises.....	186
10.8 Answer to #2.....	187
10.9 Opening files for simultaneous read/write.....	188
10.9.1 Exercises.....	188
10.10 Answer.....	189
10.11 Opening pipes.....	190
10.11.1 Exercises.....	191
10.12 Answers.....	192
10.12.1 Exercise 2.....	192
10.12.2 Exercise 3.....	192
10.13 Finding information about files.....	193
10.14 Exercises.....	195
10.15 Answers.....	196
10.15.1 Exercise 1.....	196
10.15.2 Exercise 2.....	196
10.15.3 Exercise 3.....	196
10.16 Recursing down directories.....	198
10.16.1 Exercises.....	199
10.17 Answer to Exercises.....	200
10.17.1 Exercise 1.....	200
10.17.2 Exercise 2.....	200
10.18 File locking.....	202
10.19 Buffering.....	203
10.20 Handling binary data.....	204

10.21 Best practices template for file manipulation.....	206
10.22 Chapter summary.....	207
Chapter 11: Advanced regular expressions.....	209
11.1 Assumed knowledge.....	210
11.2 Review exercises.....	211
11.3 Answers.....	212
11.3.1 Exercise 1.....	212
11.3.2 Exercise 2.....	212
11.3.3 Exercise 3.....	212
11.3.4 Exercise 4.....	212
11.4 More metacharacters.....	214
11.5 Working with multiline strings.....	215
11.5.1 Exercises.....	217
11.6 Answer.....	218
11.7 Regexp modifiers for multiline data.....	219
11.8 Backreferences.....	220
11.8.1 Special variables.....	220
11.9 Exercises.....	222
11.9.1 Advanced.....	222
11.10 Answers.....	223
11.10.1 Exercise 1.....	223
11.10.2 Exercise 2.....	223
11.10.3 Advanced Exercise 1.....	223
11.11 Section summary.....	225
Chapter 12: More functions.....	227
12.1 The grep() function.....	228
12.1.1 Exercises.....	229
12.2 Answers.....	230
12.2.1 Exercise 1.....	230
12.2.2 Exercise 2a.....	230
12.2.3 Exercise 2b.....	230
12.3 The map() function.....	231
12.3.1 Exercises.....	231
12.4 Answers.....	232
12.5 Chapter summary.....	233
Chapter 13: System interaction.....	235
13.1 system() and exec().....	236
13.1.1 Exercises.....	236

13.2 Answer.....	237
13.3 Using backticks.....	238
13.3.1 Exercises.....	239
13.4 Answers.....	240
13.4.1 Exercise 1.....	240
13.4.2 Exercise 2.....	240
13.4.3 Exercise 3.....	240
13.5 Platform dependency issues.....	241
13.6 Security considerations.....	242
13.6.1 Exercises.....	243
13.7 Answers.....	244
13.7.1 Exercise 1.....	244
13.7.2 Exercise 2.....	244
13.8 Section summary.....	246
Chapter 14: Data structures and refs.....	247
14.1 Assumed knowledge.....	248
14.2 Introduction to references.....	249
14.3 Uses for references.....	250
14.3.1 Creating complex data structures.....	250
14.3.2 Passing arrays and hashes to subroutines and functions.....	250
14.3.3 Object oriented Perl.....	250
14.4 Creating and dereferencing references.....	251
14.5 Passing multiple arrays/hashes as arguments.....	254
14.6 Complex data structures.....	255
14.7 Anonymous data structures.....	256
14.8 Exercises.....	258
14.9 Answers.....	259
14.9.1 Exercise 1.....	259
14.9.2 Exercise 2.....	260
14.10 Debugging and Persistence.....	262
14.10.1 Data::Dumper.....	262
14.10.2 Storable.....	263
14.11 YAML.....	266
14.11.1 YAML::Syck.....	267
14.11.2 YAML module.....	267
14.12 Module Exercises.....	268
14.13 Module Exercises Answers.....	269
14.14 Section summary.....	270

Chapter 15: perlstyle.....	271
15.1 perlstyle 5.8.8.....	272
Chapter 16: About databases.....	277
16.1 What is a database?.....	278
16.2 Types of databases.....	279
16.3 Database management systems.....	280
16.4 Uses of databases.....	281
16.5 Chapter summary.....	282
Chapter 17: Textfiles as databases.....	283
17.1 Delimited text files.....	284
17.1.1 Reading delimited text files.....	284
17.1.2 Searching for records.....	285
17.1.3 Sorting records.....	286
17.1.4 Writing to delimited text files.....	287
17.2 Comma-separated variable (CSV) files.....	289
17.3 Problems with flat file databases.....	290
17.3.1 Locking.....	290
17.3.2 Complex data.....	290
17.3.3 Efficiency.....	290
17.4 Chapter summary.....	291
Chapter 18: Relational databases.....	293
18.1 Tables and relationships.....	294
18.2 Structured Query Language.....	297
18.2.1 General syntax.....	297
18.2.1.1 SELECT.....	298
18.2.1.2 INSERT.....	298
18.2.1.3 DELETE.....	299
18.2.1.4 UPDATE.....	299
18.2.1.5 CREATE.....	299
18.2.1.6 DROP.....	300
18.3 Chapter summary.....	301
Chapter 19: MySQL.....	303
19.1 MySQL features.....	304
19.1.1 General features.....	304
19.1.2 Cross-platform compatibility.....	304
19.2 Comparisons with other popular DBMSs.....	305

19.2.1 PostgreSQL.....	305
19.2.2 Oracle, Sybase, etc.....	305
19.3 Getting MySQL.....	306
19.3.1 Red Hat Linux.....	306
19.3.2 Debian Linux.....	306
19.3.3 Compiling from source.....	306
19.3.4 Binaries for other platforms.....	306
19.4 Setting up MySQL databases.....	307
19.4.1 Creating the Acme inventory database.....	307
19.4.2 Setting up permissions.....	307
19.4.3 Creating tables.....	307
19.5 The MySQL client.....	310
19.6 Understanding the MySQL client prompts.....	312
19.7 Exercises.....	313
19.8 Chapter summary.....	314
Chapter 20: The DBI and DBD modules.....	315
20.1 What is DBI?.....	316
20.2 DBI documentation set.....	317
20.3 Supported database types.....	318
20.4 How does DBI work?.....	319
20.5 DBI/DBD syntax.....	320
20.5.1 Variable name conventions.....	320
20.6 Connecting to the database.....	321
20.7 Executing an SQL query.....	322
20.8 Doing useful things with the data.....	323
20.9 An easier way to execute non-SELECT queries.....	324
20.10 Quoting special characters in SQL.....	325
20.11 Exercises.....	326
20.11.1 Advanced exercises.....	326
20.12 Chapter summary.....	327
Chapter 21: DBIx::Class.....	329
21.1 Create content.....	330
Chapter 22: Acme Widget Co. Exercises.....	331
22.1 The Acme inventory application.....	332
22.2 Listing stock items.....	333
22.2.1 Advanced exercises:.....	334
22.3 Adding new stock items.....	335
22.3.1 Advanced exercises.....	335

22.4 Entering a sale into the system.....	336
22.5 Creating sales reports.....	337
22.5.1 Advanced exercises.....	337
22.6 Searching for stock items.....	338
22.6.1 Advanced exercises.....	338
Chapter 23: What is CGI?.....	339
23.1 Definition of CGI.....	340
23.2 Introduction to HTTP.....	341
23.3 Terminology.....	343
23.4 HTTP status codes.....	345
23.5 HTTP Methods.....	346
23.5.1.1 GET.....	346
23.5.1.2 HEAD.....	346
23.5.1.3 POST.....	346
23.6 Exercises.....	347
23.7 What is needed to run CGI programs?.....	349
23.8 Chapter summary.....	350
Chapter 24: Web page generation.....	351
24.1 Your public_html directory.....	352
24.2 The CGI directory.....	353
24.3 The HTTP headers.....	354
24.4 HTML output.....	355
24.5 Running and debugging CGI programs.....	356
24.5.1 Exercises.....	356
24.6 Quoting made easy.....	357
24.6.1 Here documents.....	357
24.7 Pick your own quotes.....	358
24.8 Exercises.....	359
24.9 Environment variables.....	360
24.9.1 Exercises.....	360
24.10 Chapter summary.....	361
Chapter 25: Processing form input.....	363
25.1 A quick look at HTML forms.....	364
25.2 The FORM element.....	365
25.3 Input fields.....	366
25.3.1 TEXT.....	366
25.3.2 CHECKBOX.....	366
25.3.3 SELECT.....	366

25.3.4 SUBMIT.....	366
25.4 The CGI module.....	367
25.4.1 What is a module?.....	367
25.4.2 Using the CGI module.....	368
25.4.3 Accepting parameters with CGI.....	368
25.4.4 Exercises.....	369
25.5 Practical Exercise: Data validation.....	370
25.5.1 Exercises.....	370
25.6 Practical Exercise: Multi-form "Wizard" interface.....	371
25.6.1 Exercises.....	374
25.7 Practical Exercise: File upload.....	375
25.8 Chapter summary.....	377
Chapter 26: Security issues.....	379
26.1 Authentication and access control for CGI scripts.....	380
26.1.1 Why is CGI authentication a bad idea?.....	380
26.2 HTTP authentication.....	381
26.3 Access control.....	382
26.3.1 Exercises.....	382
26.4 Tainted data.....	384
26.4.1 Exercises.....	385
26.5 cgiwrap.....	386
26.6 Secure HTTP.....	387
26.7 Chapter summary.....	388
Chapter 27: Other related Perl modules.....	389
27.1 Useful Perl modules.....	390
27.2 Failing gracefully with CGI::Carp.....	391
27.2.1 Exercise.....	392
27.3 Encoding URIs with URI::Escape.....	393
27.3.1 Exercise.....	393
27.4 Creating templates with Text::Template.....	394
27.4.1 Introduction to object oriented modules.....	394
27.4.2 Using the Text::Template module.....	394
27.4.3 Exercise.....	395
27.5 Templating with the Template Toolkit.....	396
27.6 Sending email with Mail::Mailer.....	397
27.6.1 Exercises.....	398
27.7 Chapter Summary.....	399

Chapter 28: Conclusion.....	401
28.1 What you've learned.....	402
28.2 Where to now?.....	404
28.3 Further reading -- books.....	405
28.4 The Perl home page (http://www.perl.com/).....	406
28.5 Perl Monks (http://www.perlmonks.com/).....	407
28.5.1 The Perl Monks Guide to the Monastery.....	407
28.5.1.1 Finding Your Way Around.....	407
28.5.1.1.1 Sections.....	408
28.5.1.1.2 Information.....	409
28.5.1.1.3 Find Interesting Nodes.....	409
28.5.1.1.4 Additional Miscellany.....	410
28.6 The Perl Journal (http://www.tpj.com/).....	411
28.7 Perl Mongers (http://www.pm.org/).....	412
28.8 The Richmond Perl Mongers (http://wiki.fini.net/bin/view/RichmondPM) ...	413
28.9 London Perl Mongers and NMS.....	414
28.10 O'Reilly's Perl books.....	415
28.11 Newsgroups.....	417
Chapter 29: Useful Modules.....	419
29.1 Options Processing.....	420
29.1.1 Getopt::Std.....	420
29.1.2 Getopt::Long.....	421
29.1.3 POD – plain old documentation.....	423
29.1.4 POD::Usage.....	423
29.1.5 AppConfig.....	423
29.2 File I/O.....	424
29.2.1 IO::File.....	424
29.2.2 IO::Select.....	424
29.2.3 File::Slurp.....	424
29.3 Networking.....	425
29.3.1 IO::Socket.....	425
29.3.2 Socket.....	425
29.3.3 Net::Netmask.....	425
29.3.4 Net::Ping.....	425
29.3.5 Sys::Hostname.....	425
29.4 Exercises.....	426
29.4.1 Options Processing.....	426
29.4.2 File I/O.....	426
29.4.3 Networking.....	426

Chapter 30: Packages and Creating Modules.....	427
30.1 Create content.....	428
30.2 Creating modules.....	429
30.3 Object Oriented Modules.....	430
Chapter 31: Debugging Perl.....	431
31.1 Create content.....	432
31.2 Carp module.....	433
31.3 Perl Debugger.....	434
Chapter 32: Win32.....	435
32.1 Win32::EventLog.....	436
32.1.1 Win32::EventLog Examples.....	436
32.1.2 Win32::EventLog Reference.....	437
32.1.2.1 The EventLog Object and its Methods.....	437
32.1.2.2 Other Win32::EventLog functions.....	440
32.2 Win32::NetAdmin.....	441
32.2.1 Example.....	441
32.2.2 Win32::NetAdmin provided functions.....	442
32.3 Win32::NetResource.....	448
32.3.1 Examples.....	448
32.3.2 Data Types.....	449
32.3.2.1 %NETRESOURCE.....	449
32.3.2.2 %SHARE_INFO	450
32.3.3 Functions.....	450
32.4 Win32::Service.....	453
32.4.1.1 Examples.....	453
32.4.1.2 Functions.....	454
32.5 Win32::Sound.....	455
32.5.1 Quick Sample.....	455
Chapter 33: *NIX cheat sheet.....	457
33.1 Some UNIX commands.....	458
Chapter 34: Editor cheat sheet.....	459
34.1 vi.....	460
34.1.1 Running.....	460
34.1.2 Using.....	460
34.1.3 Exiting.....	460
34.1.4 Gotchas.....	460
34.1.5 Help.....	461

34.1.6 vim.....	461
34.2 pico.....	462
34.2.1 Running.....	462
34.2.2 Using.....	462
34.2.3 Exiting.....	462
34.2.4 Gotchas.....	462
34.2.5 Help.....	462
34.3 joe.....	463
34.3.1 Running.....	463
34.3.2 Using.....	463
34.3.3 Exiting.....	463
34.3.4 Gotchas.....	463
34.3.5 Help.....	463
34.4 jed.....	464
34.4.1 Running.....	464
34.4.2 Using.....	464
34.4.3 Exiting.....	464
34.4.4 Gotchas.....	464
34.4.5 Help.....	464
Chapter 35: ASCII Pronunciation Guide.....	465
Chapter 36: HTML Cheat Sheet.....	467
Chapter 37: The Regex Coach.....	471
37.1 Abstract.....	472
37.2 Contents.....	473
37.3 Download and installation.....	474
37.3.1 Older versions, Linux, FreeBSD, Mac.....	474
37.4 Support, bug reports, mailing list.....	476
37.4.1 How to report bugs.....	476
37.5 How to use The Regex Coach.....	477
37.5.1 The main panes.....	478
37.5.2 The message areas.....	478
37.5.3 Highlighting selected parts of the match.....	478
37.5.4 The highlight buttons.....	479
37.5.5 The highlight messages.....	479
37.5.6 Walking through the target string.....	479
37.5.7 Narrowing the scan.....	480
37.5.8 The info pane.....	480
37.5.9 The parse tree.....	480

37.5.10 Replacing text.....	480
37.5.11 Splitting text.....	481
37.5.12 Single-stepping through the matching process.....	481
37.5.13 Modifiers.....	481
37.5.14 Resizing.....	482
37.5.15 Saving to and loading from files.....	482
37.5.16 Autoscroll.....	482
37.6 Known bugs and limitations.....	483
37.7 Technical information.....	484
37.7.1 Compatibility with Perl.....	484
37.8 Acknowledgements.....	485
Chapter 38: GPL2.....	487
38.1 GNU General Public License.....	488
38.1.1 Preamble.....	488
38.1.2 Terms and Conditions for Copying, Distribution and Modification.....	489
Chapter 39: Acknowledgements.....	495
39.1 Folks.....	496
39.2 Projects.....	497

Chapter 1: Overview

This chapter will...

Welcome to PerlClass.com's 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 Chapter 33 on page 471 and an editor command summary in Chapter 34 starting on page 473.

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. Lastly, an HTML cheat-sheet is provided in Chapter 36 starting on page 481 for those who need reminding.

1.2 Rough outline

- What is Perl?
- Creating and running a Perl program
- Variable types
- Operators and Functions
- Conditional constructs
- Subroutines
- Regular expressions
- File I/O
- Advanced regular expressions
- More functions
- System interaction
- References and complex data structures
- Perl Style
- Text based ("flat file") databases
- Relational database concepts and basic SQL
- The DBI and DBD modules
- Extended exercises
- CGI
- Security issues
- Other related features and Perl modules
- Win32 modules – Perl in MS Windows

1.3 Other topics we can discuss

- 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 convenient way to print out complex data structures
- Storable – convenient persistent storage of Perl data structures
- DBIx::Class – a friendly OOP-style layer on top of DBI
- Storable – persistence of complex Perl object across processes, systems, etc.
- ???
- ???
- ???

1.4 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 5.8. Perl 5 is 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, nearly all older Perl programs should work fine under Perl 5.

1.5 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.

Advanced

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

RTFM!

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

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

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".

1.6 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 back end 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 uses 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 realize that sometimes 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 5.8.9, which is available from [http://www](http://www.perl.com/) <http://www.perl.com/> or any of a number of mirror sites. Work has been moving slowly on Perl 6 and it is still early in the test stage. You can check <http://www.perl.org/> for current version status.

A Windows version is available from ActiveState (<http://www.activestate.com/>) or as part of Cygwin tool kit (<http://www.cygwin.com/>).

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.cpan.org/> and installed separately.

2.6 CPAN

CPAN is an amazing thing. It provides a comprehensive library of IT technology that can be installed quickly and used in your Perl projects.

CPAN was inspired by the Comprehensive TeX Archive Network (CTAN), but it has gone far further in organizing, indexing, mirroring, and sharing than CTAN or any other collaborative language effort. There are thousands of modules covering numerous areas.

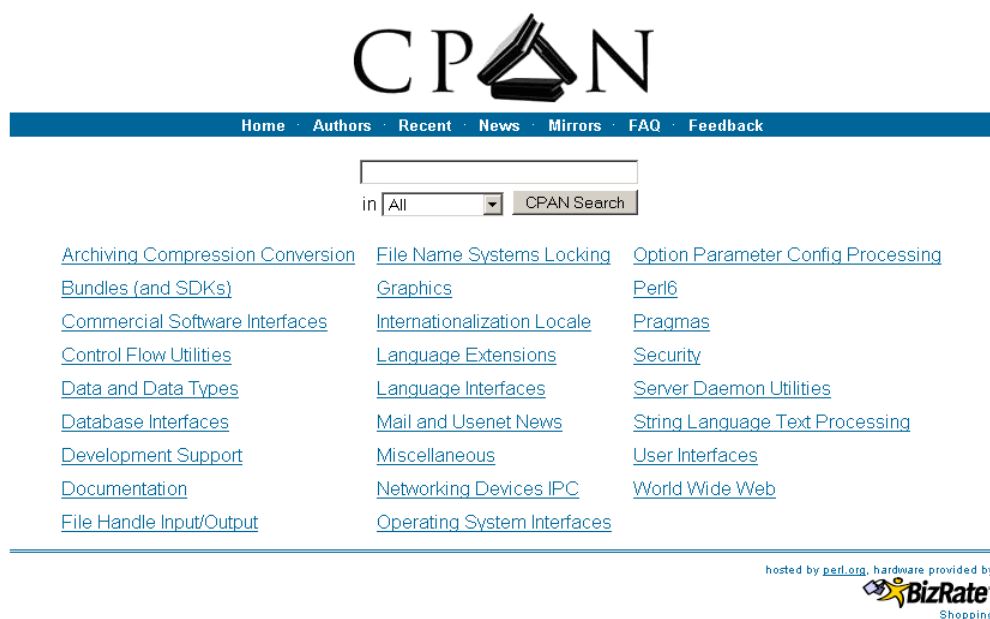


Illustration 1: <http://search.cpan.org> main page shows the main categories used by CPAN.

2.7 Slashdot

Slashdot is based on an open source Perl project known as "slash". Lots of technical news and discussion happens on slashdot:

Illustration 2: A recent sampling of <http://slashdot.org> - a Perl-based techie site.

2.8 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 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 from Windows

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	perlclass.fini.net which probably has the IP of 192.168.____.____
Your username	stu____
Your password	stu____

1. Open putty
2. Put in the hostname or IP in the host name box.
3. Before clicking open there are a few settings that are helpful to adjust.
4. Under “Translation” is a drop down for “Character set translation”. The default for this is one of the ISO8559 variants, but Red Hat derived Linuxes have been using UTF8 for many years now.
5. Under “Appearance” you can change the font size to your liking.

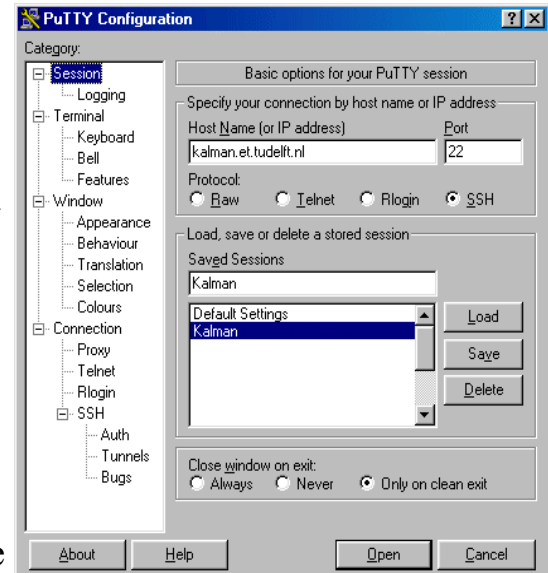


Illustration 3: putty's settings window

6. To avoid retyping your username every time you login, under “Connection” you can put in an “Auto-login username” and it will take you directly to the password prompt.
7. If you're used to X-Windows cut and paste, you will probably be more comfortable with putty following the same mouse conventions. Under “Selection” choose xterm and middle mouse will paste as it should.
8. Now return to the “Session” screen, put a helpful description under “Saved Sessions” such as “perlclass” or “Linux server for class”, and click “Save”.

9. From now on you can double click on the saved session and it will take you directly in. So double click on the saved session or click “Open”.
10. The first time you connect it will ask you to confirm the host key. Its ok to accept this for the future
11. Put in the username and password when prompted.

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 UNIX: Chapter 33 on page 471 of these notes.

Advanced

putty is deceptively powerful. There are many customizations that you can explore further through putty's documentation.

In particular, look at the key-based authentication. Whether for automating tasks or typing fewer passwords this is a powerful facility that meshes well with the ubiquitous ssh infrastructure.

The official site URL

(<http://www.chiark.greenend.org.uk/~sgtatham/putty/>) is awful to type, but there is a mirror (<http://www.putty.nl/>), and even easier, you can google for putty and the org.uk URL is the first one that comes up.

When you download putty.exe its handy to put it in C:\Windows so you can run it from command windows and the run dialog.

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] ] [ -O[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

<code>perl</code>	Perl overview (this section)
<code>perlintro</code>	Perl introduction for beginners

<code>perltoc</code>	Perl documentation table of contents
----------------------	--------------------------------------

Tutorials

<code>perlreftut</code>	Perl references short introduction
<code>perldsc</code>	Perl data structures intro
<code>perllo1</code>	Perl data structures: arrays of arrays

<code>perlrequick</code>	Perl regular expressions quick start
<code>perlretut</code>	Perl regular expressions tutorial

<code>perlboot</code>	Perl OO tutorial for beginners
<code>perltoot</code>	Perl OO tutorial, part 1
<code>perltooc</code>	Perl OO tutorial, part 2
<code>perlbot</code>	Perl OO tricks and examples

<code>perlstyle</code>	Perl style guide
------------------------	------------------

<code>perlcheat</code>	Perl cheat sheet
<code>perltrap</code>	Perl traps for the unwary
<code>perldebtut</code>	Perl debugging tutorial

<code>perlfaq</code>	Perl frequently asked questions
<code>perlfaq1</code>	General Questions About Perl
<code>perlfaq2</code>	Obtaining and Learning about Perl
<code>perlfaq3</code>	Programming Tools
<code>perlfaq4</code>	Data Manipulation
<code>perlfaq5</code>	Files and Formats
<code>perlfaq6</code>	Regexes
<code>perlfaq7</code>	Perl Language Issues
<code>perlfaq8</code>	System Interaction
<code>perlfaq9</code>	Networking

Reference Manual

<code>perlsyn</code>	Perl syntax
<code>perldata</code>	Perl data structures
<code>perlop</code>	Perl operators and precedence
<code>perlsub</code>	Perl subroutines

<code>perlfunc</code>	Perl built-in functions
<code>perlopenut</code>	Perl <code>open()</code> tutorial
<code>perlpacktut</code>	Perl <code>pack()</code> and <code>unpack()</code> tutorial
<code>perlpod</code>	Perl plain old documentation
<code>perlpodspec</code>	Perl plain old documentation format specification
<code>perlrun</code>	Perl execution and options
<code>perldiag</code>	Perl diagnostic messages
<code>perllexwarn</code>	Perl warnings and their control
<code>perldebug</code>	Perl debugging
<code>perlvar</code>	Perl predefined variables
<code>perlre</code>	Perl regular expressions, the rest of the story
<code>perlreref</code>	Perl regular expressions quick reference
<code>perlref</code>	Perl references, the rest of the story
<code>perlform</code>	Perl formats
<code>perlobj</code>	Perl objects
<code>perltie</code>	Perl objects hidden behind simple variables
<code>perldbmfilter</code>	Perl DBM filters
<code>perlipc</code>	Perl interprocess communication
<code>perlfork</code>	Perl <code>fork()</code> information
<code>perlnumber</code>	Perl number semantics
<code>perlthrtut</code>	Perl threads tutorial
<code>perlothrtut</code>	Old Perl threads tutorial
<code>perlport</code>	Perl portability guide
<code>perllocale</code>	Perl locale support
<code>perluniintro</code>	Perl Unicode introduction
<code>perlunicode</code>	Perl Unicode support
<code>perlebcdic</code>	Considerations for running Perl on EBCDIC platforms
<code>perlsec</code>	Perl security
<code>perlmod</code>	Perl modules: how they work
<code>perlmodlib</code>	Perl modules: how to write and use
<code>perlmodstyle</code>	Perl modules: how to write modules with style

<code>perlmodinstall</code>	Perl modules: how to install from CPAN
<code>perlnewmod</code>	Perl modules: preparing a new module for distribution
<code>perlutil</code>	utilities packaged with the Perl distribution
<code>perlcompile</code>	Perl compiler suite intro
<code>perlfilter</code>	Perl source filters

Internals and C Language Interface

<code>perlembed</code>	Perl ways to embed perl in your C or C++ application
<code>perldebbugs</code>	Perl debugging guts and tips
<code>perlxsut</code>	Perl XS tutorial
<code>perlxs</code>	Perl XS application programming interface
<code>perlclib</code>	Internal replacements for standard C library functions
<code>perlguts</code>	Perl internal functions for those doing extensions
<code>perlcall</code>	Perl calling conventions from C
<code>perlapi</code>	Perl API listing (autogenerated)
<code>perlintern</code>	Perl internal functions (autogenerated)
<code>perliol</code>	C API for Perl's implementation of IO in Layers
<code>perlapiio</code>	Perl internal IO abstraction interface
<code>perlhack</code>	Perl hackers guide

Miscellaneous

<code>perlbook</code>	Perl book information
<code>perltodo</code>	Perl things to do
<code>perldoc</code>	Look up Perl documentation in Pod format

<code>perlhist</code>	Perl history records
<code>perldelta</code>	Perl changes since previous version
<code>perl584delta</code>	Perl changes in version 5.8.4
<code>perl583delta</code>	Perl changes in version 5.8.3
<code>perl582delta</code>	Perl changes in version 5.8.2
<code>perl581delta</code>	Perl changes in version 5.8.1
<code>perl58delta</code>	Perl changes in version 5.8.0
<code>perl573delta</code>	Perl changes in version 5.7.3
<code>perl572delta</code>	Perl changes in version 5.7.2
<code>perl571delta</code>	Perl changes in version 5.7.1
<code>perl570delta</code>	Perl changes in version 5.7.0
<code>perl561delta</code>	Perl changes in version 5.6.1
<code>perl56delta</code>	Perl changes in version 5.6
<code>perl5005delta</code>	Perl changes in version 5.005
<code>perl5004delta</code>	Perl changes in version 5.004
<code>perlartistic</code>	Perl Artistic License
<code>perlgpl</code>	GNU General Public License

Language-Specific

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

Platform-Specific

<code>perlaix</code>	Perl notes for AIX
<code>perlamiga</code>	Perl notes for AmigaOS
<code>perlapollo</code>	Perl notes for Apollo DomainOS
<code>perlbeos</code>	Perl notes for BeOS
<code>perlbs2000</code>	Perl notes for POSIX-BC BS2000
<code>perlce</code>	Perl notes for WinCE
<code>perlcygwin</code>	Perl notes for Cygwin
<code>perldgux</code>	Perl notes for DG/UX
<code>perldos</code>	Perl notes for DOS
<code>perlepoc</code>	Perl notes for EPOC

<code>perlfreebsd</code>	Perl notes for FreeBSD
<code>perlhpx</code>	Perl notes for HP-UX
<code>perlhurd</code>	Perl notes for Hurd
<code>perlirix</code>	Perl notes for Irix
<code>perlmachten</code>	Perl notes for Power MachTen
<code>perlmacos</code>	Perl notes for Mac OS (Classic)
<code>perlmacosx</code>	Perl notes for Mac OS X
<code>perlmint</code>	Perl notes for MiNT
<code>perlmpaix</code>	Perl notes for MPE/iX
<code>perlnetware</code>	Perl notes for NetWare
<code>perlos2</code>	Perl notes for OS/2
<code>perlos390</code>	Perl notes for OS/390
<code>perlos400</code>	Perl notes for OS/400
<code>perlplan9</code>	Perl notes for Plan 9
<code>perlqnx</code>	Perl notes for QNX
<code>perlsolaris</code>	Perl notes for Solaris
<code>perltru64</code>	Perl notes for Tru64
<code>perluts</code>	Perl notes for UTS
<code>perlvmsesa</code>	Perl notes for VM/ESA
<code>perlvms</code>	Perl notes for VMS
<code>perlvos</code>	Perl notes for Stratus VOS
<code>perlwin32</code>	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`, `perlxsut`, `perlxs`, `perlcall`, `perlguits`, 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 `perllo1`.

- 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 `perldebbugs`.

- 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**, or its work-alike **nano** 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, feel free to use it, but the instructor isn't inclined to support **emacs**.

To edit a file using **pico**, type:

```
$ pico filename
```

(Note that the dollar sign is your UNIX/Linux 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 UNIX in Chapter 34 starting on page 473 in the back of these course notes, just in case you need them.

Incidentally, Chapter 35 starting on page 479 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 2 nd	3	35-38	
Camel 2 nd	6	330-337	"Switches"
Camel 3 rd	19	486-505	
perldoc	perlrun		
Cookbook 2 nd			
Learning 3 rd	2	26-27	
Learning 4 th			

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 if you're coming from a scripting mindset
- 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 2 nd	8	335-336	
Camel 2 nd	7	500	
Camel 3 rd	4	137-138	
perldoc	strict		
Cookbook 2 nd			
Learning 3 rd	B	289	
Learning 4 th			

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/perlintro/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.

RTFM!

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

4.5 More useful pragmas

It should be noted that “use” is most commonly used with loading modules. But two other pragmas are worth noting:

```
use warnings;
use diagnostics;
```

The “use warnings” statement is equivalent to the `-w` command line switch. The “use” version of things has several advantages. It is scoped instead of global so you can cut it on and off at will. A common need for this is when calling someone's code that you don't want to modify. If the code in question produces spurious and irritating warnings simply put a “no warnings” before calling the code and then restore proper etiquette with a “use warnings”. Most well-written Perl will run without spurious warnings, but sometimes something useful hasn't been written up to par and we are better off working around the limitations than rewriting it from scratch or cutting off warnings altogether.

If you're finding Perl's error messages confusing then the “use diagnostics” pragma is for you. The first time any error or warning is produced by your Perl program it will be accompanied by a paragraph or more of helpful explanation.

```
chicks $ cat diagpre
#!/usr/bin/perl -w
```

```
print $empty, "\n";
```

```
chicks $ ./diagpre Name "main::empty" used only once: possible typo at
./diagpre line 3. Use of uninitialized value in print at ./diagpre line 3.
```

```
chicks $ cat diagpost #!/usr/bin/perl -w use diagnostics; print $empty,
"\n"; chicks $ ./diagpost
```

```
Name "main::empty" used only once: possible typo at ./diagpost line 5 (#1)
```

(W once) Typographical errors often show up as unique variable names. If you had a good reason for having a unique name, then just mention it again somehow to suppress the message. The our declaration is provided for this purpose. NOTE: This warning detects symbols that have been used only once so `$c`, `@c`, `%c`, `*c`, `&c`, `sub c{}`, `c()`, and `c` (the file-handle or format) are considered the same; if a program uses `$c` only once but also uses any of the others it will not trigger this warning.

Use of uninitialized value in print at ./diagpost line 5 (#2) (w uninitialized) An undefined value was used as if it were already defined. It was interpreted as a "" or a 0, but maybe it was a mistake. To suppress this warning assign a defined value to your variables. To help you figure out what was undefined, perl tells you what operation you used the undefined value in. Note, however, that perl optimizes your program and the operation displayed in the warning may not necessarily appear literally in your program. For example, "that \$foo" is usually optimized into "that " . \$foo, and the warning will refer to the concatenation (.) operator, even though there is no . in your program.

4.6 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.

4.7 Double and single quotes

RTFM!

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/perlintro/interpolate.pl` so you can experiment with differ-

ent 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: `\\`

4.8 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/perlintro/answers/scalars.pl`.

4.9 Answers

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

```
use strict;
```

```
my $name = "Kirrily Robert";
```

```
my $street_number = 52;
```

```
my $colour = "purple";
```

```
print "My name is $name and I live in house number $street_number.\n";
```

```
print "My favourite colour is $colour.\n";
```

```
print "C:\\WINDOWS\\SYSTEM\\n";
```

```
print 'C:\\WINDOWS\\SYSTEM\\';
```

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

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

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

Advanced

Perl is full of helpful surprises. In several cases Perl will treat negative arguments as counting from the end. This works with arguments to `substr()` as well as array indexes. So `$array[-1]` will get you the last element and `@array[-10..-1]` will get you the last 10.

RTFM!

Src	Chap	Pgs	#
Nutshell 2 nd	4	65	
Camel 2 nd	2	90-91	
Camel 3 rd	3	103-104	
perldoc	perlop	..	
Cookbook 2 nd	6	199-201	
Learning 3 rd			
Learning 4 th			

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.

4.10.1 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.

4.10.2 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.

4.11 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/perlintro/answers/arrays.pl`

4.11.1 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/perlintro/answers/arrays_advanced.pl`

4.12 Answers

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

use strict;

my @friends = ("Larry", "Randal", "Tom", "Nat", "Joe");

print "First element: $friends[0]\n";
print "Last element: $friends[$#friends]\n";

print "My friends' names are @friends\n";

print "Three of my friends are @friends[1..3]\n";
```

4.12.1 Advanced Answer

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

use strict;

my @friends = ("Larry", "Randal", "Tom", "Nat", "Joe");

print "First element: $friends[0]\n";
print "Last element: $friends[$#friends]\n";

print "My friends' names are @friends\n";

print "Three of my friends are @friends[1..3]\n";

# we'll get no spaces with the following...
print @friends;
print "\n";

# set the item separator to something meaningful
$, = " and ";

# ahhh, now it works..
print @friends;
```

```
print "\n";

# print command line arguments
print "Arguments: ";
print @ARGV;
print "\n";
```

4.13 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.

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

4.13.1 Initializing 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,  
    # ...  
);
```

4.13.2 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".

4.13.3 Adding new hash elements

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

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

4.13.4 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()`.

4.13.5 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.

4.14 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/perlintro/answers/hash.pl`

4.15 Answers

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

use strict;

my %people = (
    "Larry"      => "Invented Perl",
    "Linus"      => "Invented Linux",
    "Guido"      => "Invented Python",
    "Bill"       => "Invented PC software licensing fees"
);

print "An interesting fact about Larry is: $people{'Larry'}\n";

# change someone's interesting fact
$people{"Bill"} = "wears glasses";

# add a new person
$people{"Ada"} = "invented the concept of looping in computer programs";

# what happens if we try to print someone who's not there?

print $people{"Charles"};
```

4.16 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 2 nd	4	53-59	
Camel 2 nd	2 7	127-140 403	
Camel 3 rd	28 32	653-676 884	
perldoc	perlvar English		English provides friendlier names for special variables
Cookbook 2 nd			
Learning 3 rd	3	49	\$_ quickly
Learning 4 th			

4.16.1 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".

4.16.2 `@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/perlintro/printargs.pl`.

```
#!/usr/bin/perl -w  
  
print "@ARGV\n";
```

To run the script, type:

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

You should see "foo bar baz" printed out.

4.16.3 `%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 `env` on the command line.

4.17 Exercises

1. Set `$_` to a string like "Hello, world", then print it out by using the `print()` command's default argument. (The answer is in `exercises/perlintro/answers/scalars3.pl`.)
2. 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 answer is in `exercises/perlintro/answers/argv.pl`.)
3. A user's home directory is stored in the environment variable `HOME`. Print out your own home directory. (The answer is in `exercises/perlintro/answers/env.pl`.)

4.18 Answers

4.18.1 Scalar answer

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

use strict;

$_ = "Hello, world.\n";

print;
```

4.18.2 Array Answer

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

use strict;

if (scalar @ARGV == 0) {
    print "I have no arguments.\n"
} else {
    print "My first argument is $ARGV[0].\n";
    print "My last argument is $ARGV[$#ARGV].\n"; # or $ARGV[-1]
    print "All of my arguments are @ARGV.\n";
    if (scalar @ARGV >= 4) {
        print "My 2nd to 4th arguments are @ARGV[1..3].\n";
    } else {
        print "I have less than 4 arguments.\n";
    }
}
```

4.18.3 Hash Answer

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

use strict;

print "The HOME environment variable is $ENV{'HOME'}.\n"
```

4.19 Chapter summary

- Perl variable names typically consist of alphanumeric characters and under-scores. 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 can be initialized with comma-separated scalars inside parentheses.
- 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 $items = @array;`
- Hashes are two-dimensional arrays of keys and values, and are indicated by a percent sign (%) at the beginning of the variable name.
- Hashes are initialized using a comma-separated list of scalars inside curly brackets. Whitespace and the `=>` operator (which is syntactically identical to the comma) can be used to make hash assignments look neater.
- To get the value of a hash item whose key is `foo` use `$hashname{foo}`
- Hashes have no internal order
- The `$_` variable is the default argument for many functions and operators
- The special array `@ARGV` contains all command line parameters/arguments
- 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...

RTFM!

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

5.2 Arithmetic operators

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

Table 5-5. Arithmetic operators

Operator	Example	Description
+	<code>\$a + \$b</code>	Addition
-	<code>\$a - \$b</code>	Subtraction
*	<code>\$a * \$b</code>	Multiplication
/	<code>\$a / \$b</code>	Division
%	<code>\$a % \$b</code>	Modulus (remainder when \$a is divided by \$b, e.g. <code>11 % 3 = 2</code>)
**	<code>\$a ** \$b</code>	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 slightly different things and you won't need them today (but you can read about them in the Camel if you would like to).

5.3 String operators

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

Table 5-5. String operators

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

```
my $fullname = $first_name . $mid_initial . $last_name;
my $line = '-' x 80;
my $ruler = $line . "\n";
```

5.3.1 Exercises

1. Calculate the cost of 18 widgets at \$37.00 each and print the answer (Answer: `exercises/perlintro/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/perlintro/answers/dashes.pl`)
3. Use `exercises/perlintro/operate.pl` to practice using arithmetic and string operators.

5.4 Answers

5.4.1 Exercise 1

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

use strict;

my $cost = 18 * 37;
print "The cost of 18 widgets at \$37 each is $cost.\n";
```

5.4.2 Exercise 2

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

use strict;

print "-" x 78;
```

5.4.3 Source to operate.pl

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

use strict;

# arithmetic...
print "Five times thirty is " . (5 * 30) . "\n";

# exponentiation and a foreach loop...
foreach my $entry (0..8) {
    print "2 to the power of $entry is " . 2**$entry . "\n";
}

# strings!
my $sentence = "There's more than ";
$sentence .= "one way to ";
$sentence .= "do it.";

print (($sentence . "\n") x 3);
```

5.5 File operators

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

Table 5-5. 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?

```
if (-e "~/forward") {  
    print "your email is being forwarded somewhere else";  
}
```

```
unless (-w $log_file) {  
    print "can't write to $log_file\n";  
}
```

```
if (-T "perl.exe") {  
    print "your perl.exe is a text file!\n";  
}
```

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.

Advanced

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 `perl`op and `perlfunc` manual pages and see which it appears in.

RTFM!

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

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 Chapter 8: Regular Expressions which starts on page 149.

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-sensitive functions

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

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

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

5.9 String manipulation

5.9.1 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);
```

5.9.2 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.

5.9.3 `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";               # "Goodbye"
```

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.

5.9.4 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!"
```

5.10 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)

```
my $non_negative = abs(-42);
srand(2008);      # seeding the random number generator
                  # with a constant causes the same
                  # "random" numbers each time
my $unknown = rand(100);
my $three = sqrt($nine);
```

5.11 Type conversions

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

`oct()` turns an octal number into its decimal equivalent.

`int()` truncates a number. It does not round.

`hex()` turns a hexadecimal number into its decimal equivalent.

`chr()` turns a decimal number into its character equivalent

`ord()` turns a character into its decimal equivalent

`scalar()` provides a scalar context.

```
my $fatty_decimal = hex("BEEF");  
my $secret_agent = oct(007);  
my $backspace = chr(127); # ASCII BS  
my $m = ord('m');
```

5.12 Manipulating lists and arrays

5.12.1 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 pushing items onto the end of a list and shifting them off the front.

A stack can be created by pushing items on the end of a list and popping them off.

```
# act like a stack
push(@stack,"item","item 2");
my $item = pop(@stack);

# act like a queue
push(@queue,"1","2","3","4","5","6","7","8");
my $item = shift(@queue); # get 1, 2..8 left
my $newitem = shift(@queue); # get 2, 3..8 left

push(@queue,"9","10","11"); # add three more
my $thirditem = shift(@queue); # get 3, 4..11 left

unshift(@queue,$thirditem) # put 3 back at the top of the queue
```

5.12.2 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));
```

5.12.3 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. To fully appreciate `split()` will have to wait for regular expressions, but `join` is straightforward:

```
my $glommed_thing = join(":", $user, $pass, $uid, $gid);
```

5.13 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.

```
delete $hash{getgone};

if (exists $hash{getgone}) {
    print "your Perl is sick";
}

my @keys = keys %hash;
```

5.14 Reading and writing files

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

We will cover file-related functions more in chapter 10 starting on page 173.

5.15 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.

```
$ perl -e 'print localtime(time), "\n";'
1722202710742131
$ perl -e 'print scalar localtime(time), "\n";'
Thu Aug  2 20:22:25 2007
$ perl -e 'print join(",", localtime(time)), "\n";'
46,22,20,2,7,107,4,213,1
```

The first of these produces an answer that is naturally baffling, but we'll explain it last. The second example adds only the scalar which causes the `localtime()` to be evaluated in a scalar context which leads it to produce a human-readable output. The third example omits the scalar but raps the `localtime()` in a join which separates its elements by commas. Comparing this output with the first imagine the commas disappearing and then the only difference is in the first two characters which are the seconds. So the first output is merely the list results from `localtime` concatenated together with no delimiters.

5.16 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/perlintro/answers/tmtowtdi.pl`)
2. Print a random number
3. Print a random item from an array (Answer: `exercises/perlintro/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/perlintro/answers/thirdchar.pl`)
5. Print out the date for a week ago (the answer's in `exercises/perlintro/answers/lastweek.pl`)
6. Print out a sentence in reverse
 - a. reverse the whole sentence
 - b. reverse just the words(Answer: `exercises/perlintro/answers/reverse.pl`)

5.17 Answers

5.17.1 Exercise 1

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

use strict;

my $sentence = "There's more than one way to do it.\n";

print uc($sentence);
```

5.17.2 Exercise 3

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

use strict;

my @quotes = (
    "Madness takes its toll; please have correct change.",
    "How do I set my laser printer to STUN?",
    "Why is the symbol for anarchy always written the same way?",
    "Any sufficiently advanced magic is indistinguishable from
    technology",
    "I could tell you, but then I'd have to reboot you.",
    "Real girls don't knit, they perl script.",
);

srand;          # seed the random number generator
print $quotes[rand(@quotes)] . "\n";
```

5.17.3 Exercise 4

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

use strict;

my $input = $ARGV[0] || die "You need to provide a work as an argument";

print "The third character is " . substr($input, 2, 1) . "\n";
```

5.17.4 Exercise 5

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

use strict;

my $WEEK_SECONDS = 60 * 60 * 24 * 7;

print localtime(time - $WEEK_SECONDS) . "\n";
```

5.17.5 Exercise 6

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

use strict;

my $sentence = "There's more than one way to do it.";

my $rev = reverse $sentence;
print "$rev\n";

my @words = reverse split(" ", $sentence);

print "@words\n";
```

5.18 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.**

RTFM!

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

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";

{
    my $a = "bar";          # start a new block
    print "$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.

RTFM!

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

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)
```

RTFM!

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

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
<code>==</code>	<code>\$a == \$b</code>	Equality (same as in C and other C-like languages)
<code>!=</code>	<code>\$a != \$b</code>	Inequality (again, C-like)
<code><</code>	<code>\$a < \$b</code>	Less than
<code>></code>	<code>\$a > \$b</code>	Greater than
<code><=</code>	<code>\$a <= \$b</code>	Less than or equal to
<code>>=</code>	<code>\$a >= \$b</code>	Greater than or equal to

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

Table 6-2. String comparison operators

Operator	Meaning
<code>eq</code>	Equality
<code>ne</code>	Inequality
<code>lt</code>	Less than (in "asciibetical" order)
<code>gt</code>	Greater than
<code>le</code>	Less than or equal to
<code>ge</code>	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 and 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";
}
```

RTFM!

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

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";
}
```

RTFM!

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

```
}
```

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	&&	<code>\$a && \$b</code>	True if both <code>\$a</code> and <code>\$b</code> are true; acts on <code>\$a</code> then if <code>\$a</code> is true, goes on to act on <code>\$b</code> .
or		<code>\$a \$b</code>	True if either of <code>\$a</code> and <code>\$b</code> are true; acts on <code>\$a</code> then if <code>\$a</code> is false, goes on to act on <code>\$b</code> .

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 for flow control

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 or short circuit operator:

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

What is it doing?

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

The and 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 2 nd			
Camel 2 nd	2	89	short circuit operators
Camel 3 rd	3	102	
perldoc			
Cookbook 2 nd			
Learning 3 rd	10	143	
Learning 4 th			

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 spelled differently from other language 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 `elsifunless` (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 (my $count = 0; $count <= $enough; $count++) {  
    print "Had enough?\n";  
}
```

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

```
# using a for loop  
  
for ($i = 0; $i <= $#array; $i++) {
```

```
        print $array[$i] . "\n";  
    }
```

using foreach

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

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

Advanced

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

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

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

We'll look at hash functions later.

6.7 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`

6.7.1 Answer

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

use strict;

# 1

my $unknown = 42;

if ($unknown < 3) {
    print "too small\n";
} elsif ($unknown > 7) {
    print "too big\n";
} else {
    print "just right\n";
}

# 2

my $first = "Christopher";
my $last = "Hicks";
```

```
print $first if $first lt $last;
print $last if $last gt $first;
print "\n";

# 3

my @array = ("a", "b", "c", "d", "e");

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

# 4

print "-----\n";
print "now with foreach:\n";

foreach my $element (@array) {
    print $element, "\n";
}

# 5

print "----\n";
print "hash:\n";

my %colours = (
    "red"      => "fire",
    "yellow"   => "daffodils",
    "green"    => "leaves",
    "blue"     => "ocean",
);

foreach (keys %colours) {
    print "$_ $colours{$_}\n";
}
```

6.8 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/perlintro/cat.pl`) will basically perform the same function as the UNIX **cat** command; that is, print out whatever is 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/perlintro/cat.pl < exercises/perlintro/cat.pl
```

This should make it print out its own source code.

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
}
```

or in better form

```
while (my $line = <STDIN>) {
    chomp $line; # strip trailing newline
    next unless length $line;
    last if lc($line) eq 'q';
}
```

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: Sub-routines

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.

RTFM!

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

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 Answers

7.6.1 Exercise 1

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

use strict;

sub print_first {
    my ($first) = @_;
    print "arg1=$first\n";
}

print_first("pass through","ignore","crap");
```

7.6.2 Exercise 2

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

use strict;

sub print_first_and_last {
    my ($first) = shift @_;
    my ($last) = pop @_;
    print "arg1=$first last=$last\n";
}

print_first_and_last("pass through","ignore","crap");
```

7.6.3 Exercise 3

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

use strict;

sub get_biggest_end {
    my ($first) = shift @_;
    my ($last) = pop @_;
    if ($first > $last) {
        print "$first (first) is larger than $last\n";
    } else {
```

```
        print "$last (last) is larger than $first\n";  
    }  
}  
  
print_first_and_last("pass through","ignore","crap");
```

7.7 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 capabilities, 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.

Advanced

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 it 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/perlintro/spellcheck.pl`.

8.3 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
<code>=~</code>	True if the pattern matches
<code>!~</code>	True if the pattern doesn't match

```
print "Please enter your homepage URL: ";
my $url = <STDIN>;
if ($url =~ /geocities/) {
    print "Ahhh, I see you have a geocities homepage!\n";
    # too bad it turned into a spam haven
}
```

8.4 Metacharacters

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

8.4.1 Some easy metacharacters

Table 8-2. Regular expression metacharacters

Metacharacter(s)	Matches...
<code>^</code>	Start of string
<code>\$</code>	End of string
<code>.</code>	Any single character except <code>\n</code> (though special things can happen in multiline mode)
<code>\n</code>	Newline (subtly different to <code>\$</code> - when working in multiline mode, there may be newlines embedded in the multiline string you're working with.
<code>\t</code>	Matches a tab
<code>\s</code>	Any whitespace character, such as space or tab
<code>\S</code>	Any non-whitespace character
<code>\d</code>	Any digit (0 to 9)
<code>\D</code>	Any non-digit
<code>\w</code>	Any "word" character - alphanumeric plus underscore (<code>_</code>)
<code>\W</code>	Any non-word character
<code>\b</code>	A word break - the zero-length point between a word character (as defined above) and a non-word character.

RTFM!

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

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\d/        # matches a dollar sign, then a digit,
                      # then a dot, then another digit, then
                      # another digit, eg $9.99
```

8.5 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

Some examples of quantifiers:

```

x?           # 0 or 1 "x"
x*           # 0 or more "x"
x+           # 1 or more "x"
x{5}         # exactly 5 "x"
x{5,}        # 5 or more "x"
x{5,10}      # 5-10 "x"
bore*d       # "bor", 0 or more "e", "d"
.*           # 0 or more of anything
.+           # 1 or more of anything
[ ]*=[ ]*    # match an "=" with optional spaces on either side

```

8.6 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 behavior, 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"
```

8.7 Exercises

1. What regular expression would match dollar amounts without commas and assuming that the pennies will be there.
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?

8.8 Answers

8.8.1 Exercise 1

```
/\$\d+\.\d{2}/
```

8.8.2 Exercise 2

```
/colou?r/
```

8.8.3 Exercise 3

```
/\b\w{4}\b/
```

8.9 Character classes

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

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

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

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.

8.9.1 Exercises as a group

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?

8.10 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 parentheses:

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

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

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

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

8.11 The concept of atoms

Parentheses 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 parentheses - 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,}/
```

8.12 Exercises

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

8.13 Answers

8.13.1 Exercise 1

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

use strict;

my $string = "Some text goes in here, blah blah.";
my $name = "Your Name Here";

if ($string =~ /$name/) {
    print "Your name appears in the string.\n";
} else {
    print "Your name doesn't appear in the string.\n";
}
```

8.13.2 Exercise 2

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

# call this script as ./footnote.pl < footnote.txt

while (<>) {
    s/\[[0-9a-z]\]\//g;
    print;
}
```

8.13.3 Exercise 3

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

use strict;

while (my $line = <>) {
    chomp $line;

    while ($line =~ /\t/) {
        $line =~ s/([^\t]*)\t//;
        print "$1\n";
    }
}
```

```
        if (length $line) {
            print "$line\n";
        }
    }

# this is much easier with split() as we will see shortly
while (my $line = <>) {
    chomp $line;
    my @data = split(/\t/, $line);
    foreach my $item (@data) {
        print "$item\n";
    }
}
```

8.14 `split()` function

The `split()` function provides a convenient way to take a scalar and use a regular expression to represent some definition of separator and it gives back the data between those separators. Some examples will make this seem much easier:

```
# split a sentence based on spaces
my $words = "This is a sentence.";
my @words = split(/ /,$words);
```

```
# split the time on the colons
my $time = "01:23:45";
my @timeparts = split(/:/,$time);
```

8.15 Exercises

1. Use `split()` to turn a full name into name parts.
2. Use `split()` to turn a string containing the alphabet (`$alpha="abcdefghijklmnopqrstuvwxyz"`) to produce an array containing one letter per cell.

8.16 Answers

8.16.1 Exercise 1

```
my @name_parts = split(/s+/, $name);
```

8.16.2 Exercise 2

```
my $alpha="abcedfghijklmnopqrstuvwxyz";  
my @alpha_bits = split(//, $alpha0;
```

8.17 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>;
```

RTFM!

Src	Chap	Pgs	#
Nutshell 2nd	4	78	read it now
Camel 2nd	2	53	
Camel 3rd	2	80 - 83	
perldoc	perlop		I/O Operators
Cookbook 2nd	8	300 - 302	
Learning 3rd	11	155 - 156	
Learning 4th	5	70 - 72	

`<>` is also known as the diamond operator.

- 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.

RTFM!			
Src	Chap	Pgs	#
Nutshell 2 nd	5	111	
Camel 2 nd	2	55 - 57	
Camel 3 rd	2	83 - 85	
perldoc	perlop		I/O Operators
Cookbook 2 nd	9	358 - 359	
Learning 3 rd	12	169 - 170	
Learning 4 th	12	165 - 166	

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, $_) or 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/perlinter/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/perlinter/answers/find-scripts.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/perlinter/answers/printscripts.pl`)

10.3 Answers

10.3.1 Exercise 1

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

use strict;

print "Please type something: ";
my $input = <>;
print "you typed $input"; # newline in $input
```

10.3.2 Exercise 2

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

use strict;

print "Please type something (Q to quit): ";

while (<>) {
    chomp;
    exit if lc($_) eq 'q';
    print "Please type something (Q to quit): ";
}
```

10.3.3 Exercise 3

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

use strict;

# using a while loop and angle brackets
while (<*.pl>) {
    print;
}

# using a foreach loop with the glob function
foreach (glob "*.pl") {
    print;
}
```

```
# using a named variable instead of $_
foreach my $script (glob "*.pl") {
    print $script;
}

# two even quicker methods...
print <*.pl>;
print glob "*.pl";
```

10.3.4 Advanced Exercise 1

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

use strict;

# using while and angle brackets...
while (<*.pl>) {
    open (FILE, $_) or die "Can't open file: $!";
    while (<FILE>) {
        print;
    }
}

# using foreach and the glob() function
foreach (glob "*.pl") {
    open (FILE, $_) or die "Can't open file: $!";
    while (<FILE>) {
        print;
    }
}

# using a named variable instead of $_
foreach my $script (glob "*.pl") {
    open (FILE, $script) or die "Can't open file: $!";
    while (<FILE>) {
        print;
    }
}
```

10.4 open() and friends - the gory details

10.4.1 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).

RTFM!			
Src	Chap	Pgs	#
Nutshell 2 nd	5	118 - 119	
Camel 2 nd	3	191 - 195	
Camel 3 rd	29	747 - 755	
perldoc	-f open		read it now
Cookbook 2 nd	7	247 - 252	
Learning 3 rd	11	150 - 151	
Learning 4 th	5	79 - 81	

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") or die "Can't open
/var/log/httpd/access.log: $!";
```

RTFM!

Src	Chap	Pgs	#
Nutshell 2 nd	4	55	
Camel 2 nd	2	134	
Camel 3 rd			
perl doc	perlvar		aka \$ERRNO
Cookbook 2 nd			
Learning 3 rd	11	153 - 154	
Learning 4 th	5	82 - 84	

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

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

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

close LOGFILE;

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

# print() takes an optional filehandle argument - defaults to STDOUT
print SCRIPTLOG "logfile open successful.\n"; # no comma after FH!

close SCRIPTLOG;
```

Note that you should always close a filehandle when you're finished with it. This can prevent loss of data if your script exists before buffers are written.

RTFM!

Src	Chap	Pgs	#
Nutshell 2 nd	5	138	
Camel 2 nd	3	229	
Camel 3 rd	29	808 - 809	
perldoc	-f sysopen		
Cookbook 2 nd	7	247 - 252	
Learning 3 rd			
Learning 4 th			

10.4.2 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/perlinter/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/perlinter/answers/100-count.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/perlinter/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/perlinter/answers/read-write.pl`)

10.5 Answers

10.5.1 Exercise 1

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

use strict;

open (ROMEO, "romeo.txt") or die ("Can't open romeo.txt: $!");

while (my $line = <ROMEO>) {
    print $line; # \n in data
}

close(ROMEO);
```

10.5.2 Exercise 2

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

use strict;

open (ROMEOPL, "romeo.pl") or die ("Can't open romeo.pl: $!");

while (my $line = <ROMEO>) {
    print $line unless $line =~ /\s*#/; # looks like a comment
}
```

10.5.3 Exercise 3

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

use strict;

open (COUNT, ">count.txt") or die ("Can't open count.txt: $!");

foreach (1..100) {
    print COUNT "$_\n";
}

close COUNT;
```

10.5.4 Exercise 4

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

use strict;

open (LOG, ">>log.txt") or die ("Can't open log.txt: $!");

while (<>) {
    print LOG "$_";
}

close LOG;
```

10.5.5 Exercise 5

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

use strict;

open (INFILE, "linux.txt") or die "Can't open linux.txt: $!";
open (OUTFILE, ">linux2.txt") or die "Can't write linux2.txt: $!";

while (<INFILE>) {
    print OUTFILE $_;
}

close INFILE;
close OUTFILE;
```

10.6 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 filenames in that directory accessible via functions such as `readdir()`.

RTFM!

Src	Chap	Pgs	#
Nutshell 2 nd	5	119	opendir
	5	125	readdir
Camel 2 nd	3	195	opendir
	3	202	readdir
Camel 3 rd	29	755	opendir
	29	770	readdir
perldoc	-f opendir -f readdir		
Cookbook 2 nd	9	356 - 358	
Learning 3 rd	12	171 - 173	
Learning 4 th	12	167 - 168	

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

my @files = readdir(HOMEDIR);

closedir HOMEDIR;

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

10.7 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/perlinter/answers/dirlist.pl`)

10.8 Answer to #2

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

use strict;

opendir (THISDIR, ".") || die "Can't open directory: $!";

$, = "\n";                # item separator
print sort readdir(THISDIR);

closedir THISDIR;
```

10.9 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 way, 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: $!";
foreach (@unique) {
    print OUTFILE $_;
}
close OUTFILE;
```

Advanced

Consider memory usage. If you have a 10G file, it will use at least that much memory as a Perl data structure.

10.9.1 Exercises

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

10.10 Answer

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

use strict;

open (JUNKFILE, "junk.txt") || die "Can't open junk.txt to read:
$!";

my @junk = reverse <JUNKFILE>;

close JUNKFILE;

open (JUNKFILE, ">junk.txt") || die "Can't open junk.txt to write:
$!";

foreach (@junk) {
    print JUNKFILE $_;
}

close JUNKFILE;
```

10.11 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 |") or 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") or die "Can't open pipe: $!";

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

close ROT13;
```

RTFM!

Src	Chap	Pgs	#
Nutshell 2 nd	4	59	\$
Camel 2 nd	2	130	
Camel 3 rd	28	670	
perldoc	perlvar		\$
Cookbook 2 nd	7	281 - 284	
Learning 3 rd	6	92	light
Learning 4 th			

Advanced

Make particular note of the lack of a comma after the file handle specified to print. If you accidentally put a comma there it will take the filehandle to be part of the list of items to be printed.

10.11.1 Exercises

1. Modify the second example above (provided for you as `exercises/perlinter/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/perlinter/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/perlinter/answers/string-s.pl`)

10.12 Answers

10.12.1 Exercise 2

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

use strict;

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

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

while (<INFILE>) {
    print ROT13 $_;
}

close INFILE;
close ROT13;
```

10.12.2 Exercise 3

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

use strict;

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

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

while (<INFILE>) {
    print ROT13 $_;
}

close INFILE;
close ROT13;
```

10.13 Finding information about files

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

Table 10-1. 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.
-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 2 nd	4	63 - 64	
Camel 2 nd	2	85	
Camel 3 rd	3	98	
perldoc	perlfunc		
Cookbook 2 nd			
Learning 3 rd	11	157 - 163	
Learning 4 th			

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.

10.14 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/perlinter/answers/fileexists.pl`)
2. Write a script to find zero-byte files in a directory. (Answer: `exercises/perlinter/answers/zerobyte.pl`)
3. Write a script to find the largest file in a directory: `exercises/perlinter/answers/largestfile.pl`)

10.15 Answers

10.15.1 Exercise 1

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

use strict;

print "What file should I open? ";
my $filename = <STDIN>

chomp $filename;

die "File doesn't exist" unless -e $filename;

open (IN, $filename) or die "Can't open file for reading: $!";

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

10.15.2 Exercise 2

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

use strict;

foreach (glob("*")) {
    print if -z;
}
```

10.15.3 Exercise 3

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

use strict;

my $largest_size = 0;
my $largest_filename = "";

foreach (glob("*")) {
```

```
    my $size = -s $_ ;
    if ($size > $largest_size) {
        $largest_size = $size;
        $largest_filename = $_;
    }
}

print "The largest file was $largest_filename\n";
```

10.16 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.

RTFM!

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

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. `$File::Find::dir/$_`.

10.16.1 Exercises

1. Modify the simple script above (in your scripts directory as `exercises/perlinter/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/perlinter/answers/find.pl`)

10.17 Answer to Exercises

10.17.1 Exercise 1

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

use strict;
use File::Find;

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

find (\&wanted, $dir);

sub wanted {
    return unless -T $_;
    print "$_\n";
}
```

10.17.2 Exercise 2

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

use strict;
use File::Find;

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

find (\&wanted, $dir);

sub wanted {
    return unless -T $_;
    my $filename = $_;
    open(TEXTFILE,$filename)
        or die "couldn't open $filename: $!";
    open(MORE,"|less") or die "couldn't open less/more: $!";
    while (my $line = <TEXTFILE>) {
        print MORE $line;
    }
}
```

```
        close(MORE); # or less  
        close(TEXTFILE);  
    }
```

10.18 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.

RTFM!

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

10.19 Buffering

Since the perl interpreter is written in C it relies on the C standard I/O library (stdio). Since stdio buffers our output we may run into issues where data does not end up where we expect it when we expect it to be there. Generally, writing to other programs (such as using pipes on the command) causes our output to be block buffered and writing to the screen is line buffered. Perl provides the special variable `$|` to cut off buffering for the “current” filehandle. By default `$|` is set to 0 which leaves buffering enabled. If you set it to 1 it will cut off buffering which is also known as auto-flush since every write leads to a flush of the buffer.

So to print a prompt which leaves the cursor on the line of the prompt:

```
$| = 1;  
print "prompt";
```

To choose a different filehandle:

```
my $oldfh = select(MONDO_IMPORTANTE);  
$| = 1; # cut off output buffering (autoflush)  
select($oldfh); # restore previous state
```

There is an English “shortcut”, but I don't use it. :)

?? expand this page?

10.20 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.

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

`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;
```

Advanced

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.

10.21 Best practices template for file manipulation

It's 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 read $filename: $!";

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 storing 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.

10.22 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 PerlClass.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

RTFM!

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	98 - 104	Concepts
	8	105 - 114	More
	9	115 - 127	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/perlinter/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/perlinter/answers/elvisre.pl`)
3. What pattern could be used to match a blank line? (Answer: `exercises/perlinter/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/perlinter/answers/ipre.pl`)

11.3 Answers

11.3.1 Exercise 1

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

use strict;

while (<>) {
    print if /Yasser Arafat|Boris Yeltsin|Monica Lewinsky/;
}
```

11.3.2 Exercise 2

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

use strict;

while (<>) {
    if (/Elvis (A(\.|ron|aron) )?Presley/) {
        print "That's Elvis.\n"
    } else {
        print "That's not Elvis.\n";
    }
}
```

11.3.3 Exercise 3

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

use strict;

print "Blank line.\n" if /^$/;
```

11.3.4 Exercise 4

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

```
use strict;

# a single part of the IP address looks like this...
my $pattern
    = "((2[0-4][0-9])|(25[0-5])|(1[0-9]{2})|([1-9][0-9])|([0-9]))";

while (<>) {
    if (m/^\$pattern\.$pattern\.$pattern\.$pattern$/) {
        print "IP number matched.\n";
    } else {
        print "That's not an IP number.\n";
    }
}

#
# here's how we'd actually do it
#

my @elements = split(/\./, $ipnumber);

die "Wrong number of elements" if @elements != 4;

foreach (@elements) {
    if ($_ > 255 or $_ < 0 or $_ =~ /\D/) {
        die "Element $_ is invalid";
    }
}

print "IP number is OK";
```

11.4 More metacharacters

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

Table 11-1. More metacharacters

Metacharacter	Meaning
<code>\B</code>	Match anything other than a word boundary
<code>\cX</code>	Control character, i.e. CTRL-<i>X</i>
<code>\0nn</code>	Octal character represented by <i>nn</i>
<code>\xnn</code>	Hexadecimal character represented by <i>nn</i>
<code>\l</code>	Lowercase next character
<code>\u</code>	Uppercase next character
<code>\L</code>	Lowercase until <code>\E</code>
<code>\U</code>	Uppercase until <code>\E</code>
<code>\Q</code>	quote (disable) metacharacters until <code>\E</code>
<code>\E</code>	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
```

11.5 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 em-
bodies 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 `$/` to `""` will cause input to occur in "paragraph mode", in which two or more consecutive newlines will be treated as the delimiter. Undefined `$/` will cause the entire file to be slurped in.

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

RTFM!

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

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
```

11.5.1 Exercises

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

11.6 Answer

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

use strict;

my $pattern = "Larry wall";
$/ = "\n%\n";

while (<>) {
    print if /$pattern/;
}
```

11.7 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, dot 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 11-2. Effects of single and multiline options

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

11.8 Backreferences

11.8.1 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/;
```

11.9 Exercises

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

11.9.1 Advanced

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

11.10 Answers

11.10.1 Exercise 1

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

use strict;

while (<>) {
    s(
        ^           # start of line
        (\w?)       # optional punctuation mark
        (\w+)       # first word
        (.*?)       # non-greedy match on stuff in the middle
        (\w+)       # last word
        (\w?)       # optional punctuation mark
        $
    )
    ($1$4$3$2$5)gx;
    print;
}
```

11.10.2 Exercise 2

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

use strict;

while (<>) {
    print "$&\n" if /(\w+) \1/;
}
```

11.10.3 Advanced Exercise 1

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

use strict;

$/ = ""; # suck in whole file at once
```

```
$_ = <STDIN>;    # get whole file

# this leaves linebreaks in - if you want to remove them, you'll
have
# to modify this next bit.  Or possibly the previous bit.

print "$&\n" while m/(\w+)(\s|\n)\1/g;
```

11.11 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.

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

```
# 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/perlinter/answers/grepnumber.pl`)
2. Use `grep()` to return a list of elements which are
 - a. keys to a hash (Answer: `exercises/perlinter/answers/grepkeys.pl`)
 - b. readable files (Answer: `exercises/perlinter/answers/grepfiles.pl`)

12.2 Answers

12.2.1 Exercise 1

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

use strict;

my @list = qw(2 be or not 2 be 3com 2morrow);

print grep /\d/, @list;
```

12.2.2 Exercise 2a

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

use strict;

my %hash = (
    alpha=>    "a",
    bravo=>    "b",
    charlie   =>    "c",
    delta=>    "d",
    echo  =>    "e",
);

my @array = qw(alpha zulu mary);

print grep { exists $hash{$_} } @array;
```

12.2.3 Exercise 2b

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

use strict;

my @array = qw(/etc/passwd /etc/shadow /usr/local /no/such/file);

# use -r file test operator to find readable files
print grep { -r $_ } @array;
```

12.3 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.

12.3.1 Exercises

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

12.4 Answers

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

use strict;

my @numbers = 1..20;
my @squares = map {$_**2} @numbers;

print "numbers: @numbers\n";
print "squares: @squares\n";
```

12.5 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-existent-file") or die "$!";'
cat: non-existent-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/perlinter/answers/finger.pl`)

13.2 Answer

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

use strict;

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

system("finger $username");
```

13.3 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.

RTFM!

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

13.3.1 Exercises

1. Modify your earlier finger program to use backticks instead of `system()` (Answer: `exercises/perlinter/answers/backtickfinger.pl`)
2. Change it to use `qx()` instead (Answer: `exercises/perlinter/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/perlinter/answers/qxfinger2.pl`)

13.4 Answers

13.4.1 Exercise 1

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

use strict;

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

print `finger $username`;
```

13.4.2 Exercise 2

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

use strict;

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

print qx(finger $username);
```

13.4.3 Exercise 3

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

use strict;

print qx(finger `whoami`);
```

13.5 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. 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.

The same situation used to apply to MacOS, but now that MacOS is Linux-based and tends to have much better support for free and open source software, portability has basically become a Windows versus POSIX situation. With Linux, MacOS, Solaris, and every other flavor of UNIX all living in the POSIX camp and Microsoft survives as an anomaly.

13.6 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
```

RTFM!

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

`-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.

Advanced

There is a `Safe` module available from CPAN that will let you setup sand boxes (similar to the JVM) that you can run Perl code in with arbitrary restrictions.

13.6.1 Exercises

1. Modify the `finger` program above to perform taint checking (Answer: `exercises/perlinter/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/perlinter/answers/taintfile.pl`)

13.7 Answers

13.7.1 Exercise 1

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

use strict;

$ENV{PATH} = "/usr/bin";

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

if ($username =~ /\^(\\w+)$/) {
    $username = $1;                                # $username now untainted
    system("finger $username");
} else {
    die "You're not allowed to finger $username";
}
```

13.7.2 Exercise 2

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

use strict;

print "What file do you want to output to? ";
my $filename = <STDIN>;

if ($filename =~ /\^[(-\\w.]+)$/) {
    $filename = $1;                                # $filename now untainted
} else {
    die "Bad filename in $filename";
}

open (COUNT, ">$filename") || die ("Can't open $filename: $!");

foreach (1..100) {
    print COUNT "$_\\n";
}
```

```
}
```

```
close COUNT;
```

13.8 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: Data structures and refs

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.

RTFM!

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

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/perlinter/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/perlinter/answers/exists.pl`)

14.9 Answers

14.9.1 Exercise 1

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

use strict;

my %pizza_prices = (
    "small"      => 6,
    "medium"     => 8,
    "medium"     => 10,
);

my %pasta_prices = (
    "small"      => 4,
    "medium"     => 5,
    "large"      => 7,
);

my %milkshake_prices = (
    "small"      => 2,
    "medium"     => 3,
    "large"      => 4,
);

# original, hash reference way...

my %food_prices = (
    "pizza"      => \%pizza_prices,
    "pasta"      => \%pasta_prices,
    "milkshakes" => \%milkshake_prices,
);

# and here's how we do the one with anonymous hashes

my %anon_food_prices = (
    "pizza"      => {
        "small"      => 6,
```

```

        "medium" => 8,
        "medium" => 10,
    },

    "pasta" => {
        "small" => 4,
        "medium" => 5,
        "large" => 7,
    },

    "milkshakes" => {
        "small" => 2,
        "medium" => 3,
        "large" => 4,
    },
);

# add an element...

$anon_food_prices{"salad"} = {
    "small" => 3,
    "medium" => 5,
    "large" => 7,
};

```

14.9.2 Exercise 2

```

#!/usr/bin/perl -w

use strict;

# set up some initial variables and stuff

my $scalar = "quux";

my %hash = (
    "foo" => "The first metasyntactic variable",
    "bar" => "The second metasyntactic variable",
    "baz" => "The third metasyntactic variable",
);

```

```
print "Element exists\n" if my_exists($scalar, \%hash);

sub my_exists {
    my ($scalar, $hashref) = @_;
    return 1 if exists($hashref->{$scalar});
}
```

14.10 Debugging and Persistence

14.10.1 Data::Dumper

Creating complex data structures and data driven programs can significantly simplify our Perl coding efforts. Figuring out what is wrong with your code often depends heavily on understanding the state of the data structures we are attempting to create or manipulate. The Data::Dumper module provides a convenient way to make this happen.

```
chicks$ cat datadumper
#!/usr/bin/perl -w

use strict;
use Data::Dumper;

my %anon_food_prices = (
    "pizza"      => {
        "small"   => 6,
        "medium"  => 8,
        "medium"  => 10,
    },

    "pasta"      => {
        "small"   => 4,
        "medium"  => 5,
        "large"   => 7,
    },

    "milkshakes" => {
        "small"   => 2,
        "medium"  => 3,
        "large"   => 4,
    },
);

print Dumper(\%anon_food_prices);
$ ./datadumper
```

```
$VAR1 = {
    'pasta' => {
        'large' => 7,
        'small' => 4,
        'medium' => 5
    },
    'pizza' => {
        'small' => 6,
        'medium' => 10
    },
    'milkshakes' => {
        'large' => 4,
        'small' => 2,
        'medium' => 3
    }
};
```

We passed a reference into the `Dumper()` so that the data structure would stay together. If we had passed the hash directly, the keys and values would have been turned into a list and `Dumper` would have printed out the series of individual keys and values from the main hash. The values would still be references so the inner hashes would still show up as hashes.

It is possible to use `Data::Dumper` to store and load data structures, but it is inefficient and somewhat clumsy. Storing data would simply be printing to a file-handle the output of `Dumper`. Restoring data requires loading that file and running it through “eval” which tells Perl to treat the content's of the scalar as Perl code. We'll skip examples for this because there is a better way...

14.10.2 Storable

The `Storable` module is very convenient and effective for storing Perl data structures and revivifying them. Let's simply demonstrate it:

```
chicks $ cat storeit
#!/usr/bin/perl -w

use strict;
```

```

use Storable;

my %anon_food_prices = (
    "pizza" => {
        "small" => 6,
        "medium" => 8,
        "medium" => 10,
    },
    "pasta" => {
        "small" => 4,
        "medium" => 5,
        "large" => 7,
    },
    "milkshakes" => {
        "small" => 2,
        "medium" => 3,
        "large" => 4,
    },
);

store \%anon_food_prices, 'food_prices.storable';
chicks $ ./storeit
chicks $ ls -l food_prices.storable
-rw-r--r-- 1 chicks users 161 Feb 18 01:29 food_prices.storable
chicks $ cat retrieveit
#!/usr/bin/perl -w

use strict;
use Storable;
use Data::Dumper;

my $prices = retrieve 'food_prices.storable';

print Dumper($prices);

chicks@aroundafraid 01:30:05 src $ ./retrieveit
$VAR1 = {
    'pasta' => {

```

```
        'small' => 4,  
        'large' => 7,  
        'medium' => 5  
    },  
    'pizza' => {  
        'small' => 6,  
        'medium' => 10  
    },  
    'milkshakes' => {  
        'small' => 2,  
        'large' => 4,  
        'medium' => 3  
    }  
};
```

So a reference turns into a file and back into a reference without any data loss. The “storable” files are binaries that aren't portable to other processor architectures unless you use `nstore()` and `nretrieve()`.

Note that our retrieval program didn't turn the reference back into a named hash. For the purposes needed a reference was the most convenient handle to hold onto the data with.

14.11 YAML

YAML, short for YAML Ain't Markup Language, is a data serialization language which provides a human-readable and somewhat editable format for storing structured data. JSON provides a similar functionality with different priorities. YAML is a superset of JSON.

Lists are represented by lines starting with dashes. Key/value pairs (hashes in Perl-speak) are represented as a key followed by a colon then the corresponding value. Let's look at some examples.

A list:

- apple
- orange
- grape

A hash:

```
sum: 900
avg: 100
cnt: 9
```

A hash of arrays:

```
apple:
  - granny smith
  - red delicious
orange:
  - fresh
  - from concentrate
grape:
  - green
  - not green
```

And so on.

More details can be found at <http://yaml.org> and in the module documentation for the following modules.

14.11.1 YAML::Syck

For quick and straightforward YAML reading and writing you can use `YAML::Syck`. This module is a Perl wrapper around `libsyck` which is a YAML 1.0 compliant parser and writer. The synopsis, slightly edited, from the `perldoc` is pretty straightforward:

```
use YAML::Syck;

$data = Load($yaml);
$yaml = Dump($data);

# $file can be an IO object, or a filename
$data = LoadFile($file);
DumpFile($file, $data);

# A string with multiple YAML streams in it
$yaml = Dump(@data);
@data = Load($yaml);
```

14.11.2 YAML module

In the great spirit of TIMTOWTDI, we have another module for parsing and producing YAML. The `YAML` module is a pure Perl implementation of YAML, and thus isn't as zippy as `YAML::Syck`. In fact it is recognized by the author to be about 10 times slower. The `Load()`, `Dump()`, `LoadFile()` and `DumpFile()` functions are still provided as with `YAML::Syck`, but there is also an object-oriented interface.

14.12 Module Exercises

1. Create a hash of hashes and print it out with `Data::Dumper`.
2. Store your hash in a file with `Storable`.
3. Create a script to read in the storable file and print it out.
4. Extend the previous script to modify the data structure (by adding or changing) and save it back out. Bonus: make it interactive.
5. Create a script to read in your storable file and save it as YAML.
6. Modify your read/modify/rewrite script to use YAML formatted files instead of `Storable` files.

14.13 Module Exercises Answers

Forthcoming. :)

14.14 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: perlstyle

In this chapter...

We will learn what it means to be stylish in Perl.

15.1 perlstyle 5.8.8

What follows is the perlstyle page from the Perl 5.8.8 distribution. It raises a number of points worth considering while developing in Perl.

Each programmer will, of course, have his or her own preferences in regards to formatting, but there are some general guidelines that will make your programs easier to read, understand, and maintain.

The most important thing is to run your programs under the `-w` flag at all times. You may turn it off explicitly for particular portions of code via the `no warnings` pragma or the `$^W` variable if you must. You should also always run under `use strict` or know the reason why not. The `use sigtrap` and even `use diagnostics` pragmas may also prove useful.

Regarding aesthetics of code lay out, about the only thing Larry cares strongly about is that the closing curly bracket of a multi-line BLOCK should line up with the keyword that started the construct. Beyond that, he has other preferences that aren't so strong:

- 4-column indent.
- Opening curly on same line as keyword, if possible, otherwise line up.
- Space before the opening curly of a multi-line BLOCK.
- One-line BLOCK may be put on one line, including curlies.
- No space before the semicolon.
- Semicolon omitted in "short" one-line BLOCK.
- Space around most operators.
- Space around a "complex" subscript (inside brackets).
- Blank lines between chunks that do different things.
- Uncuddled elses.
- No space between function name and its opening parenthesis.
- Space after each comma.
- Long lines broken after an operator (except `and` and `or`).
- Space after last parenthesis matching on current line.
- Line up corresponding items vertically.
- Omit redundant punctuation as long as clarity doesn't suffer.

Larry has his reasons for each of these things, but he doesn't claim that everyone else's mind works the same as his does.

Here are some other more substantive style issues to think about:

- Just because you *CAN* do something a particular way doesn't mean that you *SHOULD* do it that way. Perl is designed to give you several ways to do anything, so consider picking the most readable one. For instance

```
open(FOO,$foo) || die "Can't open $foo: $!";
```

is better than

```
die "Can't open $foo: $!" unless open(FOO,$foo);
```

because the second way hides the main point of the statement in a modifier. On the other hand

```
print "Starting analysis\n" if $verbose;
```

is better than

```
$verbose && print "Starting analysis\n";
```

because the main point isn't whether the user typed `-v` or not.

Similarly, just because an operator lets you assume default arguments doesn't mean that you have to make use of the defaults. The defaults are there for lazy systems programmers writing one-shot programs. If you want your program to be readable, consider supplying the argument.

Along the same lines, just because you *CAN* omit parentheses in many places doesn't mean that you ought to:

```
return print reverse sort num values %array;  
return print(reverse(sort num (values(%array))));
```

When in doubt, parenthesize. At the very least it will let some poor schmuck bounce on the `%` key in `vi`.

Even if you aren't in doubt, consider the mental welfare of the person who has to maintain the code after you, and who will probably put parentheses in the wrong place.

- Don't go through silly contortions to exit a loop at the top or the bottom, when Perl provides the `last` operator so you can exit in the middle. Just "outdent" it a little to make it more visible:

```
LINE:  
    for (;;) {  
        statements;
```

```

        last LINE if $foo;
        next LINE if /^#/;
        statements;
    }

```

- Don't be afraid to use loop labels--they're there to enhance readability as well as to allow multilevel loop breaks. See the previous example.
- Avoid using `grep()` (or `map()`) or 'backticks' in a void context, that is, when you just throw away their return values. Those functions all have return values, so use them. Otherwise use a `foreach()` loop or the `system()` function instead.
- For portability, when using features that may not be implemented on every machine, test the construct in an `eval` to see if it fails. If you know what version or patchlevel a particular feature was implemented, you can test `$] ($PERL_VERSION in English)` to see if it will be there. The `Config` module will also let you interrogate values determined by the **Configure** program when Perl was installed.
- Choose mnemonic identifiers. If you can't remember what mnemonic means, you've got a problem.
- While short identifiers like `$gotit` are probably ok, use underscores to separate words in longer identifiers. It is generally easier to read `$var_names_like_this` than `$varNamesLikeThis`, especially for non-native speakers of English. It's also a simple rule that works consistently with `VAR_NAMES_LIKE_THIS`.

Package names are sometimes an exception to this rule. Perl informally reserves lowercase module names for "pragma" modules like `integer` and `strict`. Other modules should begin with a capital letter and use mixed case, but probably without underscores due to limitations in primitive file systems' representations of module names as files that must fit into a few sparse bytes.

- You may find it helpful to use letter case to indicate the scope or nature of a variable. For example:

<code>\$ALL_CAPS_HERE</code>	constants only (clashes with perl vars!)
<code>\$Some_Caps_Here</code>	package-wide global/static
<code>\$no_caps_here</code>	function scope <code>my()</code> or <code>local()</code> variables

Function and method names seem to work best as all lowercase. E.g., `$obj->as_string()`.

You can use a leading underscore to indicate that a variable or function should not be used outside the package that defined it.

- If you have a really hairy regular expression, use the `/x` modifier and put in some whitespace to make it look a little less like line noise. Don't use slash as a delimiter when your regexp has slashes or backslashes.
- Use the new `and` and `or` operators to avoid having to parenthesize list operators so much, and to reduce the incidence of punctuation operators like `&&` and `||`. Call your subroutines as if they were functions or list operators to avoid excessive ampersands and parentheses.
- Use here documents instead of repeated `print()` statements.
- Line up corresponding things vertically, especially if it'd be too long to fit on one line anyway.

```

$IDX = $ST_MTIME;
$IDX = $ST_ETIME      if $opt_u;
$IDX = $ST_CTIME      if $opt_c;
$IDX = $ST_SIZE       if $opt_s;

mkdir $tmpdir, 0700 or die "can't mkdir $tmpdir: $!";
chdir($tmpdir)      or die "can't chdir $tmpdir: $!";
mkdir 'tmp', 0777 or die "can't mkdir $tmpdir/tmp: $!";

```

- Always check the return codes of system calls. Good error messages should go to `STDERR`, include which program caused the problem, what the failed system call and arguments were, and (VERY IMPORTANT) should contain the standard system error message for what went wrong. Here's a simple but sufficient example:


```

      opendir(D, $dir)      or die "can't opendir $dir: $!";
      
```
- Line up your transliterations when it makes sense:


```

      tr [abc]
        [xyz];
      
```
- Think about reusability. Why waste brainpower on a one-shot when you might want to do something like it again? Consider generalizing your

code. Consider writing a module or object class. Consider making your code run cleanly with `use strict` and `use warnings` (or `-w`) in effect. Consider giving away your code. Consider changing your whole world view. Consider... oh, never mind.

- Try to document your code and use Pod formatting in a consistent way. Here are commonly expected conventions:
 - use `C<>` for function, variable and module names (and more generally anything that can be considered part of code, like filehandles or specific values). Note that function names are considered more readable with parentheses after their name, that is `function()`.
 - use `B<>` for commands names like `cat` or `grep`.
 - use `F<>` or `C<>` for file names. `F<>` should be the only Pod code for file names, but as most Pod formatters render it as italic, Unix and Windows paths with their slashes and backslashes may be less readable, and better rendered with `C<>`.
- Be consistent.
- Be nice.

Perl style is important, but it could be conveyed better than the perlstyle page manages. Rewrite this and mention perltidy.

Chapter 16: About databases

In this chapter...

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

16.1 What is a database?

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

16.2 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.

16.3 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.

16.4 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.

16.5 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 17: 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.

17.1 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. Any printable character has a chance of appearing in your data. This tends to make tab-delimited files much more convenient than the various alternatives. Comma-delimited files are particularly painful since it often puts strings in quotes to allow for commas inside of 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`.

17.1.1 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.

17.1.2 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;

```

17.1.3 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: $!";

my @records;

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.

17.1.4 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";
```

17.2 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)

17.3 Problems with flat file databases

17.3.1 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.

17.3.2 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.

17.3.3 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.

17.4 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 18: 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.

18.1 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 18-1. 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 18-2. 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).

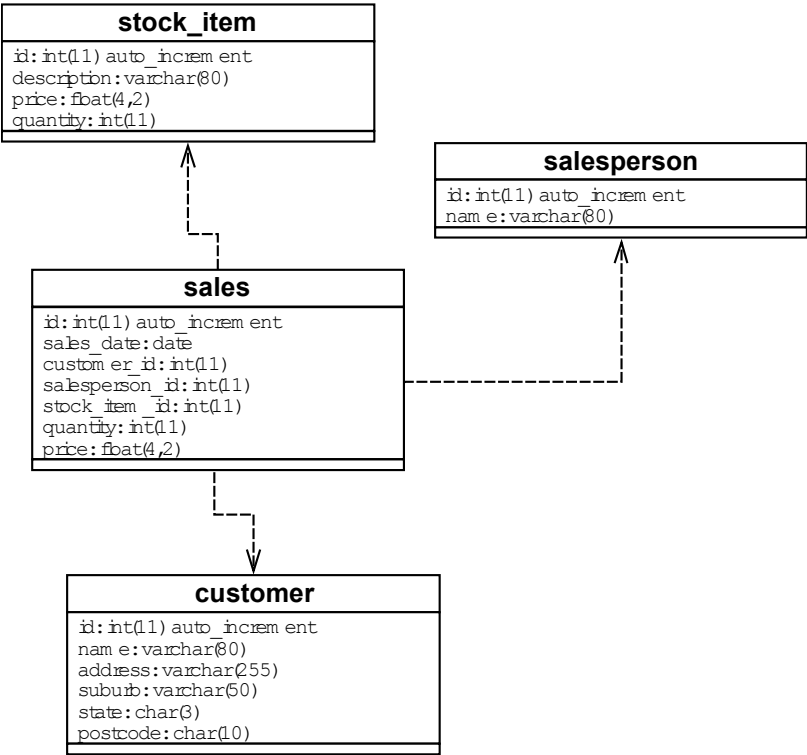


Illustration 18-4: UML-style ERD of the example schema

Table 18-3. the stock item table

stock_item
<i>id</i>
description
price
quantity

Table 18-4. the customer table

customer
<i>id</i>
name
address
suburb
state

postcode

Table 18-5. the salesperson table

salesperson
<i>id</i>
name

Table 18-6. the sales table

sales
<i>id</i>
sale date
salesperson id
customer id
stock item id
quantity
price

18.2 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://dev.mysql.com/doc/>.

18.2.1 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 18-7. 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 parentheses 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.

18.2.1.1 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.

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

```
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
```

18.2.1.2 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 interact-

ive use, however it is very bad practice to omit them in programs. *Always* specify column names in `insert` statements.

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

```
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.

18.2.1.3 DELETE

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

```
delete from tablename where condition
```

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

18.2.1.4 UPDATE

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

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

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

18.2.1.5 CREATE

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

```
create table tablename (
    column coltype options,
    column coltype options,
    ...
    primary key (colname)
)
```

Data types include (but are not limited to):

Table 18-8. Some data types

INT	an integer number
FLOAT	a floating point number
CHAR(<i>n</i>)	character data of exactly <i>n</i> characters
VARCHAR(<i>n</i>)	character data of up to <i>n</i> 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.

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

18.2.1.6 DROP

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

```
drop table tablename
```

```
drop table contactlist
```

18.3 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 19: 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.

19.1 MySQL features

19.1.1 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

19.1.2 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

19.2 Comparisons with other popular DBMSs

19.2.1 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/>

19.2.2 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/>

19.3 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:

19.3.1 Red Hat Linux

Download the appropriate RPM packages, and type `rpm -i package-name.rpm`

MySQL is included with Fedora, Red Hat Enterprise, CentOS, and any other current Red Hat-derived Linuxes. So the standard package installers should have no trouble installing this for you. For instance;

```
# yum install mysql
```

19.3.2 Debian Linux

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

19.3.3 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`.

19.3.4 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.

19.4 Setting up MySQL databases

Out of date material removed...to be filled in with current info

19.4.1 Creating the Acme inventory database

Out of date material removed...to be filled in with current info

19.4.2 Setting up permissions

Out of date material removed...to be filled in with current info

Table 19-2. 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

19.4.3 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, e.g. `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)  
);
```

19.5 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`.

```
$ 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)

19.6 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:

```
'>
```

```
">
```

19.7 Exercises

1. Connect to a database which has the same name as your login (for instance, `stu01`) by typing **`mysql -p stu01`** (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.

19.8 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 20: 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.

20.1 What is DBI?


Like the Perl modules discussed throughout this course, the DBI and DBD modules are written by Perl people and distributed for 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.

RTFM!

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

20.2 DBI documentation set

CPAN

[Home](#) · [Authors](#) · [Recent](#) · [News](#) · [Mirrors](#) · [FAQ](#) · [Feedback](#)

in All

[Tim Bunce](#) > DBI-1.53

DBI-1.53

This Release

DBI-1.53

[\[Download\]](#) [\[Browse\]](#)

01 Nov 2006

Other Releases

DBI-1.52 — 08 Aug 2006

[Goto](#)

Links

[\[CPAN Testers \]](#) [\[Discussion Forum \]](#) [\[View/Report Bugs \(5\) \]](#) [\[Tools \]](#)

CPAN Testers

PASS (85) [\[View \]](#)

Rating

★★★★★ (27) [\[View \]](#) [\[Rate this distribution \]](#)

License

Unknown

Special Files

[MANIFEST](#) [META.yml](#) [Makefile.PL](#) [README](#)


Modules

Bundle::DBI	A bundle to install DBI and required modules.	11.03
DBD::DBM	a DBI driver for DBM & MLDBM files	0.03
DBD::ExampleP		11.12
DBD::File	Base class for writing DBI drivers	0.35
DBD::NullP		11.04
DBD::Proxy	A proxy driver for the DBI	0.2004
DBD::Sponge	Create a DBI statement handle from Perl data	11.10
DBI	Database independent interface for Perl	1.53
DBI::Const::GetInfo::ANSI	ISO/IEC SQL/CLI Constants for GetInfo	1.03
DBI::Const::GetInfo::ODBC	ODBC Constants for GetInfo	1.03
DBI::Const::GetInfoReturn	Data and functions for describing GetInfo results	1.04
DBI::Const::GetInfoType	Data describing GetInfo type codes	1.05
DBI::DBD	Perl DBI Database Driver Writer's Guide	11.22
DBI::DBD::Metadata	Generate the code and data for some DBI metadata methods	1.05
DBI::FAQ	The Frequently Asked Questions for the Perl5 Database Interface	0.38
DBI::Profile	Performance profiling and benchmarking for the DBI	1.07
DBI::ProfileData	manipulate DBI::ProfileDumper data dumps	1.0
DBI::ProfileDumper	profile DBI usage and output data to a file	1.0
DBI::ProfileDumper::Apache	capture DBI profiling data from Apache/mod_perl	1.1
DBI::ProfileSubs	Subroutines for dynamic profile Path	
DBI::ProxyServer	a server for the DBD::Proxy driver	0.3005
DBI::SQL::Nano	a very tiny SQL engine	0.03

Documentation

DBI::Changes	List of significant changes to the DBI
DBI::PurePerl	a DBI emulation using pure perl (no C/XS compilation required)
DBI::Roadmap	Planned Enhancements for the DBI
DBI::W32ODBC	An experimental DBI emulation layer for Win32::ODBC
TASKS	Want to help? These things need doing...
Win32::DBIODBC	Win32::ODBC emulation layer for the DBI
dbiprof	command-line client for DBI::ProfileData
dbiproxy	A proxy server for the DBD::Proxy driver

hosted by [perl.org](#), hardware provided by

 [Shopping](#)

20.3 Supported database types

Databases supported by Perl's DBI module include:

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

20.4 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 -xvzf DBI.tar.gz`**, then follow the instructions in the **README** file distributed with the module.

20.5 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.

20.5.1 Variable name conventions

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

Table 20-1. DBI module variable naming conventions

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

20.6 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 connection which returns a database handle we can use
my $dbh = DBI->connect($dsn, $username, $password);

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

20.7 Executing an SQL query

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

# execute it
$sth->execute() or 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();
```

20.8 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";
}
```

Advanced

Of the above methods, `fetchrow_array()` is the only one that does not require an understanding of Perl references. References are not a beginner-level topic, but for those who are interested, they are documented in chapter 4 of the Camel. They are worth learning if only for the added benefit of being able to access fields by name when using the `fetchrow_hashref` method.

20.9 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.

20.10 Quoting special characters in SQL

Sometimes you want to use a value in your SQL which may contain characters which have special behavior 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);
```

20.11 Exercises

1. Use `exercises/perl/db/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.

20.11.1 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!

20.12 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 21: DBIx::Class

In this chapter...

You will learn about the Object-Relational Model module
`DBIx::Class`.

21.1 Create content

This is still a **work in progress**, this will be here eventually

Chapter 22: 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.

22.1 The Acme inventory application

In your `exercises/perl/db/` 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.

22.2 Listing stock items

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

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

my $driver = 'mysql';
my $database = 'stuXX';
my $username = 'stuXX';
my $password = 'your_password_here';

my $dsn = "DBI:$driver:database=$database";
my $dbh = DBI->connect($dsn, $username, $password)
    or 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

22.2.1 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

22.3 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/perl/db/acme/addstock.pl`

22.3.1 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

22.4 Entering a sale into the system

1. The program `exercises/perl/db/acme/sale.pl` provides an interface which can be used to input data pertinent to the occurrence 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.

22.5 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.

22.5.1 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.

22.6 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

22.6.1 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 23: 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.

23.1 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.

23.2 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 Chapter 36 starting on page 481.

RTFM!

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

23.3 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.

23.4 HTTP status codes

Table 23-1. 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

23.5 HTTP Methods

23.5.1.1 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.

23.5.1.2 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.

23.5.1.3 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

23.6 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. 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 '^['.
```

2. 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*.
3. 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.
4. 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`
 5. 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` (we don't have permission to read)
 - `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 dif-

ferent URL)

- `ENCRYPT /index.html HTTP/1.0` (a method which isn't known to our server)

23.7 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.

23.8 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 24: Web page generation

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.

24.1 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. Of course, you will need to replace both the hostname and username to match your specific setup. So if your login name is `stu03` and you are using the PerlClass.com training server at `perlclass.fini.net`, you can access your web pages at `http://perlclass.fini.net/~stu03`.

24.2 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. On the PerlClass.com training server the directory name will be simply `cgi` for convenience of typing and to differentiate it from the system-wide `cgi-bin`.

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` directory.
3. Change to your `cgi` 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` directory is world executable - note that it is not compulsory to have a `cgi` 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

24.3 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`

24.4 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` directory as `hello.cgi`.

24.5 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 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/httpd`, 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.

24.5.1 Exercises

1. Look at the output of the `hello.cgi` script by pointing a web browser (such as Netscape) at `http://hostname/~stuXX/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.

24.6 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.

24.6.1 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.

24.7 Pick your own quotes

Another way of avoiding excessive backslashes in your code is to use the `qq()` or `q()` operators/functions.

RTFM!			
Src	Chap	Pgs	#
Nutshell 2 nd	4	46	
Camel 2 nd	2	41	
Camel 3 rd	2	63 - 64	
perldoc	perlop		
Cookbook 2 nd	1	3	
Learning 3 rd	3	43 - 44	qw()
Learning 4 th			

```
print qq(\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>`.

24.8 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 `` element) using a `foreach` loop. If you need help with HTML, there's a cheat sheet in Chapter 36 starting on page 481.
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`

24.9 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).

24.9.1 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 spelled 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.

24.10 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 25: 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.

25.1 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 Chapter 35 starting on page 479 of these notes, but here's a brief run-down of the major parts of HTML forms:

25.2 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 25-1. 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.

25.3 Input fields

Some of the input fields you can use in your form include:

25.3.1 TEXT

A text input field `<INPUT TYPE="TEXT" NAME="email_address">`

25.3.2 CHECKBOX

Creates a yes/no checkbox. Saying CHECKED will make it checked by default.

```
<INPUT TYPE="CHECKBOX" NAME="send_email" CHECKED>
```

25.3.3 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="cooking">Cookery</OPTION>
  <OPTION VALUE="reading">Reading</OPTION>
  <OPTION VALUE="bushwalking">Bushwalking</OPTION>
</SELECT>
```

25.3.4 SUBMIT

Creates a button which, when pressed, will submit the form.

```
<INPUT TYPE="SUBMIT" VALUE="Press me!">
```

25.4 The `CGI` module

25.4.1 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.

RTFM!			
Src	Chap	Pgs	#
Nutshell 2 nd	10	376 - 398	
Camel 2 nd			
Camel 3 rd			
perldoc	CGI		
Cookbook 2 nd		756 - 791	
Learning 3 rd			
Learning 4 th			

25.4.2 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;
```

25.4.3 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!";
```

25.4.4 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.

25.5 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";
}
```

25.5.1 Exercises

1. Open the form for the validation program in your browser. Try submitting the form with various inputs.

25.6 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" NAME="..." VALUE="...">
```

Source code for this example is available as `cgi/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 air-
speed
        velocity of an unladen swallow is $swallow.
        </p>
    );
}
```

25.6.1 Exercises

1. Add another question to the wizard.cgi script.

25.7 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>";
```

25.8 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 26: 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.

26.1 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.

26.1.1 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.

26.2 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 realizes 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 recognize as meaning that the authentication failed, and display the body.

26.3 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.

To create an `.htpasswd` file we will utilize the `htpasswd` command like so:

```
$ htpasswd -c .htpasswd chicks
New password:
Re-type new password:
Adding password for user chicks
```

The `-c` option is necessary the first time you run this command so that it will create the `.htpasswd` file. Subsequent invocations for this username or other usernames don't require the `-c` option.

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 /home/stuXX/.htpasswd
require valid-user
```

This authentication will apply to the directory in which the `.htaccess` file is placed and any subdirectories.

26.3.1 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?

26.4 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
```

RTFM!

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

-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.

26.4.1 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/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.

26.5 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.

26.6 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.

26.7 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 by 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 27: 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.

27.1 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.cpan.org/>) and installed either using the CPAN module distributed with Perl, or by following the steps described in the README file distributed with each module.

27.2 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 'fatalToBrowser';
```

RTFM!

Src	Chap	Pgs	#
Nutshell 2 nd	8	192	
Camel 2 nd	7	385	
Camel 3 rd	32	878	
perldoc	Carp		
Cookbook 2 nd	12	473 - 475	
Learning 3 rd			
Learning 4 th			

27.2.1 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

27.3 Encoding URIs with `URI::Escape`

Sometimes we want to output anchor tags `` 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);
```

27.3.1 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`

27.4 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.

27.4.1 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 27-1. Attributes and Methods of a cup

Object	Attributes	Methods
Cup	<ul style="list-style-type: none"> • colour • handle (does it have one?) • contents (water, coffee, etc) • fullness 	<ul style="list-style-type: none"> • 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.

27.4.2 Using the `Text::Template` module

Like the cup, our text template has attributes and methods.

Table 27-2. Attributes and Methods of Text::Template

Text::Template	<ul style="list-style-type: none"> • TYPE - the type of template it is, eg a file, a string you created earlier, etc • SOURCE - the file-handle or variable name in which the template can be found 	<ul style="list-style-type: none"> • 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.tpl'
});
```

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.

27.4.3 Exercise

1. Type `perldoc Text::Template` and look at the documentation for this module
2. `cgi/letter.cgi` implements the example above. Examine the source code.
3. Make some changes to the letter template and see if they work.

27.5 Templating with the Template Toolkit

This is more standard and flexible and useful than the module discussed in the previous section, but **the content needs to be written to cover it.**

27.6 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.

27.6.1 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

27.7 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 28: Conclusion

In the conclusion...

Summing up and various paths for further study.

28.1 What you've learned

Now you've completed PerlClass.com, 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
- 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
- What CGI is
- How HTTP allows web user agents (browsers) to communicate with web servers and retrieve 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
- 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

28.2 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/>)

28.3 Further reading -- books

- Alligator Descartes & Tim Bunce, "Programming the Perl DBI", O'Reilly and Associates, 2000
- Randy Jay Yager, 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.

REALLY?
perl.com
 THE SOURCE FOR PERL

0 Really Network
[perl.com](#)
[Perl Books](#)
[Perl Courses](#)
[Perl Tutorials](#)
[Perl Podcasts](#)

0 Really Network
[perl.com](#)
[Perl Books](#)
[Perl Courses](#)
[Perl Tutorials](#)
[Perl Podcasts](#)

Join perl.com Today | View Site

Perl.com

Perl.com includes resources on [downloading and installing Perl](#), a [six-part tutorial on learning Perl](#), the [Catalist web application framework](#), and hundreds of other articles and resources. Both new and experienced programmers refine their skills and contribute to the worldwide Perl community.

CPAN Review

CPAN Module Review: XML::Atom
 Monday December 13, 2005 3:49PM

I recently needed to filter and process some Atom feeds. I know enough XML that I could process them with my own SAX filter, but this seemed like a better opportunity to use the XML::Atom module. Fortunately, it was very easy. chromatic

[More CPAN Reviews](#)

Perl Weblog Posts

Perl Jobs
 Friday February 16, 2007 6:01AM

Competition in the marketplace is a good thing right? So now we have both job openings and jobs to apply for. I'm looking for a Perl programmer - so let's see which one of them is most useful. Currently the [read more](#) Dave Cross

Grants: Calls for Proposals
 Friday February 02, 2007 12:22PM

If you have an idea for doing some work for the Perl community and you think it's worth a grant, please send your grant entry to [t2f-proposals@perl-foundation.org](#). Submission deadline is the last day of February, voting starts in March. [read more](#) Curtis Roe

London Needs Perl Programmers
 Friday January 26, 2007 4:35PM

Dave Cross just posted a short analysis of Perl programmers in London and the job situation there. This matches what I've heard, and what I noticed when I was in Europe last summer. There's plenty of work available for people. [read more](#) chromatic

Why Do You Contribute to Community Documentation?
 Friday January 26, 2007 4:35PM

It's important to understand volunteer motivation to encourage further altruistic and mutually beneficial behavior. O'Reilly Editor Andy Oram has created a short survey for people to contribute to community documentation. Do you answer questions on mailing lists about how to? [read more](#) chromatic

[More O'Reilly Posts](#)

Articles

Advanced HTML::Template: Widgets
 by Philip Taylor

HTML::Template is a templating module for HTML, made powerful by its simplicity. Its minimal set of operators enforces a strict separation between presentation and logic. However, sometimes that minimalism makes templates unfriendly. Philip Taylor demonstrates how to reuse templates smaller than an entire page--and how this simplifies your applications.

[More HTML::Template Articles](#)

Unitless Windows Module Installation with PPM
 by Josh Brostrom

Perl and Unix-like systems often come with compilers and make utilities. Windows systems rarely do. Installing Perl modules on Windows can be somewhat difficult by hand. Fortunately, Andrew's Perl::Win32 module does much of the pain, and it's highly customizable too. Josh Brostrom demonstrates how to install Perl modules with PPM and how to create your own repositories.

[More Unitless Windows Articles](#)

Using Java Classes in Perl
 by Andrew Hanenkamp

Java has a huge amount of standard libraries and APIs. Some of them don't have Perl counterparts yet. Fortunately, using Java classes from Perl is easy--with Inline::Java. Andrew Hanenkamp shows you how.

[More Using Java Classes Articles](#)

[View the archive](#)

Perl Recipe of the Day

You want the standard Exporter module to define the external interface to your module.

[Do it now](#)

[Perl Cookbook](#)

Site Feeds

Subscribe to perl.com's RSS feed:
[New to RSS?](#)

Related Books

Perl Best Practices
 by Damian Conway
 July 2005 \$29.95 USD

[More books](#)

Perl Jobs

[Perl Jobs](#)

[View All Jobs](#)
[Join a Job](#)

O'Reilly Learning Center

[Perl for CGI Programming](#)

In this course, participants will learn not only CGI, but the the computer language Perl. This language is generally regarded as the most useful computer language for processing and manipulating text based data.

[More](#)

Articles

HTML::Template: Fragments
 by Philip Taylor

HTML::Template is a templating module for HTML, made powerful by its simplicity. Its minimal set of operators enforces a strict separation between presentation and logic. However, sometimes that minimalism makes templates unfriendly. Philip Taylor demonstrates how to reuse templates smaller than an entire page--and how this simplifies your applications.

[More HTML::Template Articles](#)

Unitless Windows Module Installation with PPM
 by Josh Brostrom

Perl and Unix-like systems often come with compilers and make utilities. Windows systems rarely do. Installing Perl modules on Windows can be somewhat difficult by hand. Fortunately, Andrew's Perl::Win32 module does much of the pain, and it's highly customizable too. Josh Brostrom demonstrates how to install Perl modules with PPM and how to create your own repositories.

[More Unitless Windows Articles](#)

Using Java Classes in Perl
 by Andrew Hanenkamp

Java has a huge amount of standard libraries and APIs. Some of them don't have Perl counterparts yet. Fortunately, using Java classes from Perl is easy--with Inline::Java. Andrew Hanenkamp shows you how.

[More Using Java Classes Articles](#)

[View the archive](#)

Perl Recipe of the Day

You want the standard Exporter module to define the external interface to your module.

[Do it now](#)

[Perl Cookbook](#)

Site Feeds

Subscribe to perl.com's RSS feed:
[New to RSS?](#)

Related Books

Perl Best Practices
 by Damian Conway
 July 2005 \$29.95 USD

[More books](#)

Perl Jobs

[Perl Jobs](#)

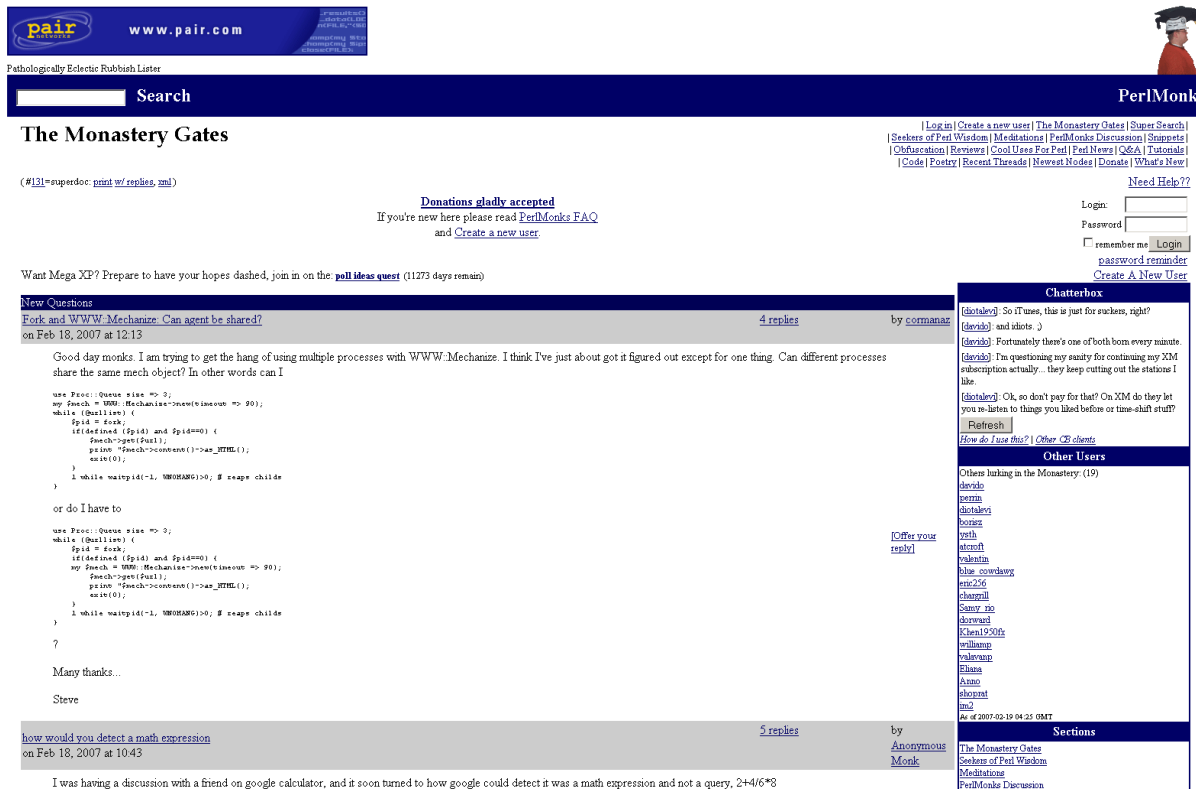
[View All Jobs](#)
[Join a Job](#)

O'Reilly Learning Center

[Perl for CGI Programming](#)

In this course, participants will learn not only CGI, but the the computer language Perl. This language is generally regarded as the most useful computer language for processing and manipulating text based data.

28.5 Perl Monks (<http://www.perlmonks.com/>)



Pathologically Eclectic Rubbish Lister

Search

PerlMonks

[Login](#) | [Create a new user](#) | [The Monastery Gates](#) | [Super Search](#) | [Sisters of Perl Wisdom](#) | [Meditations](#) | [PerlMonks Discussion](#) | [Stagnate](#) | [Chit-chat](#) | [Reviews](#) | [Cool Uses For Perl](#) | [Perl News](#) | [Q&A](#) | [Tutorials](#) | [Code](#) | [Poetry](#) | [Recent Threads](#) | [Newest Nodes](#) | [Donate](#) | [What's New](#)

Need Help??

Login:
 Password:
☐ remember me [Login](#)
[password reminder](#)
[Create A New User](#)

Want Mega XP? Prepare to have your hopes dashed, join in on the: [perl ideas quest](#) (11273 days remain)

New Questions

[Fork and WWW::Mechanize: Can agent be shared?](#) 4 replies by [gormanar](#)
on Feb 18, 2007 at 12:13

Good day monks I am trying to get the hang of using multiple processes with WWW::Mechanize. I think I've just about got it figured out except for one thing. Can different processes share the same mech object? In other words can I

```
use PProc:Queue size => 2;
my $mech = WWW::Mechanize->new(timeout => 90);
while ($mech->is_alive) {
    $pid = fork;
    if ($pid == 0) {
        $mech->get(Purl);
        print "Fetched->content()->as_HTML();";
        exit(0);
    }
    $mech->waitpid($pid, 0);
}

or do I have to

use PProc:Queue size => 2;
while ($mech->is_alive) {
    $pid = fork;
    if ($pid == 0) {
        $mech = WWW::Mechanize->new(timeout => 90);
        $mech->get(Purl);
        print "Fetched->content()->as_HTML();";
        exit(0);
    }
    $mech->waitpid($pid, 0);
}

?
```

Many thanks...

Steve

[how would you detect a math expression](#) 5 replies by [Anonymous Monk](#)
on Feb 18, 2007 at 10:43

I was having a discussion with a friend on google calculator, and it soon turned to how google could detect it was a math expression and not a query, 2+4/6*8

Chatterbox

[dshalev](#): So iTunes, this is just for each one, right?
[dshalev](#): and slots :)
[dshalev](#): Fortunately there's one of both here every minute
[dshalev](#): I'm questioning my sanity for continuing my XM subscription actually... they keep cutting out the stations I like.
[dshalev](#): Ok, so don't pay for that? On XM do they let you re-listen to things you liked before or time-shift stuff?
[Refresh](#)
[How do I use this?](#) | [Other CP clients](#)

Other Users

Others lurking in the Monastery: (19)

[dshalev](#)
[perlin](#)
[dshalev](#)
[booz](#)
[perh](#)
[stern](#)
[valentin](#)
[rhu](#)
[champl](#)
[Sany](#)
[dshalev](#)
[Kant12300](#)
[william](#)
[valentin](#)
[Elana](#)
[Auro](#)
[dshalev](#)
[m](#)

as of 2007-02-18 04:25 GMT

Sections

[The Monastery Gates](#)
[Sisters of Perl Wisdom](#)
[Meditations](#)
[PerlMonks Discussion](#)

28.5.1 The Perl Monks Guide to the Monastery

Welcome to Perl Monks, the Monastery of Perl. We hope your stay is long and enjoyable. You are probably wondering what this is all about. Hopefully this page will answer some of those questions.

In the words of different people, Perl Monks is:

- a medium for making Perl as non-intimidating to learn and as easy to use as possible;
- a place for Perl programmers (such as you) to improve your skills and share your expertise;
- a community which allows everyone to grow and learn from each other.

28.5.1.1 Finding Your Way Around

The Monastery has a number of areas, called "Sections", where you can read and contribute to discussions in a threaded messageboard-like forum format. There are also other useful reposi-

ories of information which will assist you in your Perl and Perl Monks endeavors.

28.5.1.1.1 Sections

[Seekers of Perl Wisdom](#) - The place you can go when you have got a question on how to do something or are unsure why something just isn't working. Then other Perl Monks can offer you their wisdom and suggestions.

[Meditations](#) - Have you found out something amazing about Perl that you just need to share with everyone. Have you had a Perl epiphany, or found something in Perl that just blows your mind. This is the place for those neat little tricks and amazing discoveries.

[PerlMonks Discussion](#) - For discussions relating specifically to this web site, and how things work around here. For example, if you think the Monastery could be improved in some particular way, raise it for discussion here.

[Categorized Questions and Answers](#) - Our own ever-growing compendium of "frequently asked" Perl-related questions and their answers. If you're faced with a problem and your inclination is to think "I'm sure this has been solved a thousand times before", then check here before you go posting to [Seekers of Perl Wisdom](#).

[Tutorials](#) - An ever-growing online textbook from which you can learn the basics of Perl or some groovy stuff that you haven't tried before. This area is managed by the [Pedagogues](#).

[Obfuscated code](#) - Got code that it would take a Perl grand master to understand? Put it here so we can stare at it in awe after we've run it and found out what it does.

[Perl Poetry](#) - The name pretty much says it all.

[Cool Uses for Perl](#) - Have you automated a part of your life that wouldn't have been possible without the power of Perl? Are you using Perl to do something unique and humorous that you're convinced no one else has thought of? Tell us about it!

[Snippets Section](#) - Have you written something clever that is incredibly useful, but hard to write the first time? Add it here so people can benefit and learn from it.

[Code Catacombs](#) - The place to put your full-blown programs and scripts that others might find useful.

[Reviews](#) - If you are shopping around for the Perl module or book which is just right for your needs, read these reviews — written by your fellow Perl Monks — to help you make an informed decision. Conversely, if you have used a module or read a book, and you think other Perl Monks might benefit from your experiences, please share them here by writing a review!

[Perl News](#) - Relevant news and announcements from the Perl Community. Pulls together items from sources such as [use Perl](#) and [O'Reilly](#).

28.5.1.1.2 Information

The [PerlMonks FAQ](#) - Your one-stop shop for Nearly Everything You Ever Wanted To Know About PerlMonks. Maintained by the [SiteDocClan](#).

[Tidings](#) - *aka* What's New at PerlMonks.

[Voting/Experience System](#) - Many newcomers are confused by this aspect of PerlMonks. This should clear things up.

[Perl FAQ](#) and **[Library](#)** - Our local copy of the standard Perl documentation set. Note, however, that the content is not being maintained and is now a couple versions old.

[Outside Links](#) - Various other sites that Perl Monks might find useful. Note, however, that this has been superseded by [Where can I find more information on...](#). See especially [Perl-Monks-Related Resources on Other Servers](#).

28.5.1.1.3 Find Interesting Nodes

[The Monastery Gates](#) - The "default" page of the web site, it shows recent nodes from all sections which have been deemed most worthy of exposure — the "face" of PerlMonks.

[Super Search](#) - Full-text and title searches, with additional filtering by section, age, author, and many other criteria.

[Newest Nodes](#) - An up-to-the minute listing of all the nodes which were created since "the last time you checked".

[Recently Updated Home Nodes](#) - Similar to [Newest Nodes](#), a listing of the user homenodes which have been modified since "the last time you checked". (Note that only changes designated by their authors as "significant" will register in this list.)

[Recently Active Threads](#) - A threaded view of the Monastery's active content. It's like [Newest Nodes](#) on steroids.

[Selected Best Nodes](#) - A random selection of 50 of the top 2000 nodes, as ranked by node [reputation](#). The selection is re-sampled daily.

[Best Nodes](#) - The top 10 nodes of the Day, the Week, and the Month, and the top 20 nodes of the Year.

[Worst Nodes](#) - The bottom 10 nodes of the Day, the Week, and the Month, and the bottom 20 nodes of the Year.

[Saints in our Book](#) - "Saints" here is more figurative, or honorary; this is a list of Monks who have at least 3000 [experience points](#), which technically makes them [Level 13: Curate](#), not

[Level 26: Saint.](#)

28.5.1.1.4 Additional Miscellany

[**The St. Larry Wall Shrine**](#) - A neat compendium of articles by and about the creator of Perl.

[**Offering Plate**](#) - If you find this place to be of value, you can show your appreciation by helping defray the costs of keeping the site up and running.

[**Awards**](#) - Accolades and other noteworthy public mentions of PerlMonks.

[**Craft \(deprecated\)**](#) - This was a place for perlsmiths to showcase their code. New submissions should go in the [Code Catacombs](#) section, but [Craft](#) still makes an interesting read.


[**Buy Stuff**](#) - Yes, you can actually buy Perl and PerlMonks related gear... such as a t-shirt with the famous [camel code](#) obfu on it!

28.6 The Perl Journal (<http://www.tpj.com/>)

Founded in 1996 by Jon Orwant, *The Perl Journal* was published through January 2006, and was the leading publication for and about Perl Programming.

28.7 Perl Mongers (<http://www.pm.org/>)

Many localities have formed Perl Mongers groups to help encourage more users of Perl and to let Perl fans socialize.



Perl Mongers

[Contact](#)
[FAQ](#)
[Invite a guru](#)
[Running a group](#)
[Start a group](#)

User groups

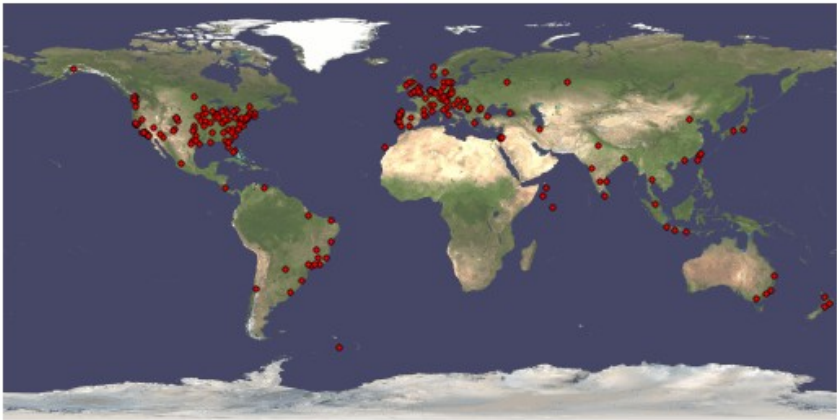

[Africa](#)
[Asia](#)
[Central America](#)
[Europe](#)
[Non-geographical](#)
[North America](#)
[Oceania](#)
[South America](#)

Affiliates

[mod_perl](#)
[Perl advocacy](#)
[Perl training](#)
[Perl.com](#)
[Perlmonks](#)
[Search CPAN](#)
[The Perl Foundation](#)
[Use Perl](#)
[YAPC](#)

Perl Mongers

Perl Mongers is a loose association of international Perl User Groups.

Copyright © 2006 The Perl Foundation.
The Perl camel image is a trademark of [O'Reilly Media, Inc.](#) Used with permission.

28.8 The Richmond Perl Mongers (<http://wiki.fini.net/bin/view/RichmondPM>)

The Richmond Perl Mongers are the closest group that meets regularly.

The screenshot shows the Richmond Perl Mongers Twiki page. The page layout includes a header with the FINI logo and navigation links. The main content area is divided into sections: "Richmond Perl Mongers", "Meetings", "Resources", "Articles", and "Historic". The "Meetings" section lists a meeting on March 2, 2007. The "Resources" section links to a discussion list. The "Articles" section lists Perl hacks and testing modules. The "Historic" section lists earlier meetings from 2007 back to 2006. On the right side, there is a sidebar with a "RichmondPM" header, a "Hello Christopher" message, a "Log Out" link, and a "My links" section with various links like "Work", "OutHouse", "UUPF Wiki", etc. At the bottom of the page, there is a copyright notice and a Twiki logo.

Illustration 5: The Richmond Perl Mongers Home Page

The Richmond.PM site is powered by Twiki.

28.9 London Perl Mongers and NMS

The London Perl Mongers have a group project known as NMS which provides an alternative set of scripts for the Matt's Script Archive scripts which had become quite outdated. <http://nms-cgi.sourceforge.net/> is the official site for the rewritten scripts. The improvement is so drastic and desperately needed that Matt's Script Archive site itself refers new users to the NMS scripts.

28.10 O'Reilly's Perl books

O'Reilly is the leading technical publisher of books about Perl, and many other wonderful Internet technologies. Their Perl books are written by core folks who have developed the language.

The screenshot shows the O'Reilly website's Perl book category page. The header includes the O'Reilly logo, a navigation menu (Home, Network, Store, Safari Books Online, Conferences, Courses, Academic Solutions, About), and a search bar. Below the header is a red banner with links: Complete List, Bestsellers, New Releases, Upcoming Titles, By Publisher, By Series, Out of Print, and Order Info.

The main content area is titled "Perl" and features a list of books under the "PUBLICATION DATE" tab. The books listed are:

- Mastering Regular Expressions, Third Edition** by Jeffrey E. F. Friedl, August 2006, \$44.99 USD. Description: Written in the lucid, entertaining tone that makes a complex, dry topic become crystal-clear to programmers, and sprinkled with solutions to complex real-world problems, Mastering Regular Expressions, 3rd Edition, offers a wealth of... [Read more.](#)
- Building Tag Clouds in Perl and PHP** by Jim Bumgardner, May 2006, \$9.99 USD. Description: Tag clouds are everywhere on the web these days. First popularized by the web sites Flickr, Technorati, and del.icio.us, these amorphous clumps of words now appear on a slew of... [Read more.](#)
- Perl Hacks** by chromatic, Damian Conway, Curtis Poe, May 2006, \$29.99 USD. Description: Perl Hacks taps into the collective wisdom of the world's most creative Perl gurus, so you can learn from their experiences. It's the perfect book for experienced developers looking for... [Read more.](#)
- Intermediate Perl** by Randal L. Schwartz, brian d foy, Tom Phoenix, March 2006, \$39.99 USD. Description: Perl programmers need a clear roadmap for improving their skills. Intermediate Perl teaches a working knowledge of Perl's objects, references, and modules -- all of which makes the language so... [Read more.](#)
- Wicked Cool Perl Scripts** by Steve Oualine, February 2006, \$29.95 USD.

On the left side, there is a "TOPICS" menu with links to various categories: Business & Culture, Databases, Design & Graphics, Digital Audio & Video, Digital Photography, Hardware, Home & Office, Networking & Sys Admin, Operating Systems, Programming (with sub-links for .NET & Windows Programming, Ajax, C#, C/C++, Certification, Games, Java, Other Programming, Perl, PHP, Project & Career Management, Python, Ruby, Secure Programming, Visual Basic, Web Services, XML, Science & Math, Security, Software Engineering, and The Web), and International Sites.

On the right side, there are promotional banners:

- Buy Direct and Save:** Buy 2 Books, Get the 3rd FREE! Use discount code "opc10". All orders over \$29.95 qualify for free shipping within the US.
- Radar Web 2.0 Report:** Web 2.0 Principles and Best Practices -- What does Web 2.0 mean for your company? Get the latest on the why, what, who, and how of Web 2.0 in this O'Reilly Radar Report. [Read more.](#)
- Short Cuts:** Good. Fast. Cheap. O'Reilly Short Cuts. PDF documents on cutting edge topics. Focused information in an easy-to-use, portable package. New titles include:
 - CompTIA A+ Essentials 220-601 Exam Guide
 - Lead Generation on the Web
 - What's New in Apache Web Server 2.2?[View all Short Cuts](#)
- Local Bookstores:**

Illustration 6: <http://www.oreilly.com/pub/topic/perl>

O'REILLY®

Home Network Store Safari Books Online Conferences Courses Academic Solutions About

Complete List | Bestsellers | New Releases | Upcoming Titles | By Publisher | By Series | Out of Print | Order Info

Perl

PUBLICATION DATE UPDATING **BESTSELLING** ALPHABETICAL

Learning Perl, Fourth Edition
By Randal L. Schwartz, Tom Phoenix, Brian d foy
July 2005
\$39.95 USD
Informed by their years of success at teaching Perl as consultants, the authors have re-engineered the Llama to better match the pace and scope appropriate for readers getting started with... [Read more.](#)

Programming Perl, Third Edition
By Larry Wall, Tom Christiansen, Jon Orwant
July 2000
\$49.95 USD
Programming Perl is not just a book about Perl; it is also a unique introduction to the language and its culture, as one might expect only from its authors. This... [Read more.](#)

Mastering Regular Expressions, Third Edition
By Jeffrey E. F. Friedl
August 2006
\$44.99 USD
Written in the lucid, entertaining tone that makes a complex, dry topic become crystal-clear to programmers, and sprinkled with solutions to complex real-world problems, Mastering Regular Expressions, 3rd Edition, offers a wealth of... [Read more.](#)

Perl Cookbook, Second Edition
By Tom Christiansen, Nathan Torkington
August 2003
\$49.95 USD
Find a Perl programmer, and you'll find a copy of Perl Cookbook nearby. Perl Cookbook is a comprehensive collection of problems, solutions, and practical examples for anyone programming in Perl... [Read more.](#)

Regular Expression Pocket Reference
By Tony Stubbins
August 2003
\$9.95 USD
Ideal as an introduction for beginners and a quick reference for advanced programmers, Regular Expression Pocket Reference is a comprehensive guide to regular expression APIs for C, Perl, PHP, Java... [Read more.](#)

Perl Pocket Reference, Fourth Edition
By John Vromans
July 2002
\$9.95 USD
The Perl Pocket Reference, 4th Edition provides a complete overview of the Perl programming language, all packed into a convenient, carry-around booklet. It is updated for Perl 5.8, and covers... [Read more.](#)

Perl Best Practices
By Damian Conway
July 2005
\$39.95 USD
Perl Best Practices offers a collection of 256 guidelines on the art of coding to help you write better Perl code—in fact, the best Perl code you possibly can. The... [Read more.](#)

Perl in a Nutshell, Second Edition
By Stephen Spinhour, Ellen Siever, Nathan Patwardhan
June 2002
\$39.95 USD
This complete guide to Perl includes the basics of the programming language itself, plus CGI programming, XML processing, network programming, database interaction, and graphical user interfaces. The expanded second edition... [Read more.](#)

Intermediate Perl
By Randal L. Schwartz, Brian d foy, Tom Phoenix
March 2006
\$39.99 USD
Perl programmers need a clear roadmap for improving their skills. Intermediate Perl teaches a working knowledge of Perl's objects, references, and modules -- all of which makes the language so... [Read more.](#)

Beginning Perl for Bioinformatics
By James Tisdell
October 2001
\$39.95 USD
This book shows biologists with little or no programming experience how to use Perl, the ideal language for biological data analysis. Each chapter focuses on solving a particular problem or... [Read more.](#)

Perl Hacks
By chromatic, Damian Conway, Curtis Poe
May 2006
\$29.99 USD
Perl Hacks taps into the collective wisdom of the world's most creative Perl gurus, so you can learn from their experiences. It's the perfect book for experienced developers looking for... [Read more.](#)

Buy Direct and Save
Buy 2 Books, Get the 3rd FREE!
Use discount code "spc10"
All orders over \$29.95 qualify for **free shipping** within the US.

Radar Web 2.0 Report
Web 2.0 Principles and Best Practices -- What does Web 2.0 mean for your company? Get the latest on the why, what, who, and how of Web 2.0 in this O'Reilly Radar Report. [Read more.](#)

Short Cuts
Good. Fast. Cheap. O'Reilly Short Cuts
PDF documents on cutting edge topics. Focused information in an easy-to-use, portable package.
New titles include:
• [CompTIA A+ Essentials 220-601 Exam Guide](#)
• [Lead Generation on the Web](#)
• [What's New in Apache Web Server 2.2](#)
[View all Short Cuts](#)

Local Bookstores
Team O'Reilly (US/CA) and Club O'Reilly (International) are stores who have joined in partnership with O'Reilly to ensure plentiful stock of current and earlier titles.

INTERNATIONAL SITES

[UK](#)
[France](#)
[Germany](#)
[Japan](#)

About O'Reilly | Contact | Jobs | Press Room | How to Advertise | Privacy Policy

© 2007, O'Reilly Media, Inc.
All trademarks and registered trademarks appearing on oreilly.com are the property of their respective owners.

Illustration 7: The most significant O'Reilly Perl books

28.11 Newsgroups

Some people only know these as google groups:

- comp.lang.perl.announce newsgroup
- comp.lang.perl.moderated newsgroup
- comp.lang.perl.misc newsgroup

Chapter 29: Useful Modules

In this chapter...

We'll show you how to save time and make your programs more consistent using a few Perl modules

29.1 Options Processing

Two modules are commonly used for options processing – `Getopt::Long` and `Getopt::Std`. Both are built into Perl and so can be assumed to be almost anywhere Perl runs. The interface for `Getopt::Long` is cleaner and it isn't limited to single character options.

29.1.1 `Getopt::Std`

`Getopt::Std` provides an easy way to single character command line flags and set Perl variables. When used with strict-mode these variables need to be declared with “our”. Once you've used the module you will be able to call `getopt()` with a string containing the letters of your valid options. So if you wanted to take the as e, i, or o you would do something like:

```
modules $ cat optstd
#!/usr/bin/perl -w

use strict;
use Getopt::Std;

our($opt_e,$opt_i,$opt_o);

getopt('eio');

print "got e ($opt_e)\n" if $opt_e;
print "got i ($opt_i)\n" if $opt_i;
print "got o ($opt_o)\n" if $opt_o;
modules $ ./optstd -e foo
got e (foo)
modules $ ./optstd -e foo -o bar
got e (foo)
got o (bar)
modules $ ./optstd -e foo -o bar -i baz
got e (foo)
got i (baz)
```

```
got o (bar)
```

One limitation clear from using this script is that it cannot deal with flag-style options. `Getopt::Std` also includes the slightly more useful `getopts()` function which accepts boolean flags and options with arguments. The string of letters of valid options is again the first argument. If any of those have colons after them they are as options with arguments rather than as simple flag options which is the default without the colon. So if we wanted `e` and `o` to be options with arguments and `i` to be a boolean option:

```
modules $ cat optstds
#!/usr/bin/perl -w

use strict;
use Getopt::Std;

our($opt_e,$opt_i,$opt_o);

getopts('e:io:');

print "got e ($opt_e)\n" if $opt_e;
print "got i ($opt_i)\n" if $opt_i;
print "got o ($opt_o)\n" if $opt_o;
modules $ ./optstds -e asdjkl -i
got e (asdjkl)
got i (1)
modules $ ./optstds -e asdjkl -i -o baz
got e (asdjkl)
got i (1)
got o (baz)
```

In an effort to simplify things in a post-strict world you can pass a hash reference as a second argument to `getopt()` or `getopts()`.

29.1.2 Getopt::Long

`Getopt::Long` is a much nicer way to process command line arguments. It allows for “gnu-style” long arguments. You get to choose the variable that an op-

tion sets so it makes sense when it is used. It also validates numeric arguments. When you call `GetOptions()` you should pass it a series of paired items. The first half of each pair is a string listing the name(s) of the option and ending with “=i” for numeric, “=s” for string or nothing for flags. The second half of each pair is a reference to a variable where the value of the argument can be stored.

So to take “v”, “verbose”, “i”, and “f” as options with i taking an integer and f taking a string argument:

```
modules $ cat longopt
#!/usr/bin/perl -w

use strict;
use Getopt::Long;

my $verbose = 0;
my $file = '/dev/null';
my $increment = 1;

GetOptions(
    "v|verbose" => \$verbose,
    "f=s" => \$file,
    "i=i" => \$increment,
);

print "our values are verbose=$verbose inc=$increment file=$file\n";
modules $ ./longopt
our values are verbose=0 inc=1 file=/dev/null
modules $ ./longopt -v
our values are verbose=1 inc=1 file=/dev/null
modules $ ./longopt -v -i 99
our values are verbose=1 inc=99 file=/dev/null
modules $ ./longopt -v -f /etc/shadow
our values are verbose=1 inc=1 file=/etc/shadow
```

`Getopt::Long` includes a number of other handy features including negatable options and multiple arguments.

29.1.3 POD – plain old documentation

[Thanks wikipedia...]

As the title indicates POD is short for plain old documentation. Perl makes it very convenient to include documentation in your script and automatically skips any documentation while running it. POD is designed to be a simple, clean language with just enough syntax to be useful. It purposefully does not include mechanisms for fonts, images, colors or tables. Some of its goals are:

- * Easy to parse
- * Easy to convert to other languages, such as HTML or TeX
- * Easy to incorporate sample code
- * Easy to read without a POD formatter (i.e. in its source-code form)
- * Easy to write in – otherwise programmers won't write the documentation!

Although the author of `perlpod` notes that "The Pod format is not necessarily sufficient for writing a book", books have in fact been written in an extended version of POD; this special version included formatting codes for tables and footnotes, and is used by O'Reilly & Associates to produce several Perl books, most notably *Programming Perl* by Larry Wall, Tom Christiansen, and Jon Orwant. A slightly extended, modified version of pod, called mod, was used to write *Higher-Order Perl*, by Mark Jason Dominus.

29.1.3.1 Use

POD is the language used for most documentation in the Perl world. This includes Perl itself, nearly all publicly-released modules, many scripts, most design documents, many articles on Perl.com and other Perl-related web sites, and the Parrot virtual machine.

POD is rarely read in the raw, although it is designed to be readable without the assistance of a formatting tool. Instead, it is read with the `perldoc` tool, or converted into Unix man pages (`pod2man`) or Web-standard HTML pages (`pod2html`).

Pure POD files usually have the extension `.pod`, but `pod` is mostly used directly in Perl code, which typically uses the `.pl` and `.pm` extensions. (The Perl interpreter's parser is designed to ignore `pod` in Perl code.) In source code files, the documentation is generally placed after the `__END__` marker (which also helps syntax highlighting in some editors to display it as comments).

29.1.3.2 pod formatting details

POD files are written in an ASCII-compatible encoding, such as Latin-1 or Unicode. A `pod` parser always assumes that the file it is parsing doesn't start with `pod`; it ignores all lines until it sees a `pod` directive. `pod` directives must come at the beginning of a line, and all begin with an equal sign. The POD parser will then assume that all following lines are POD, until it encounters a line consisting of the `=cut` directive. Any content following that is ignored until the parser encounters another POD directive. Thus, POD can be intermixed with executable source code if the language's parser knows how to recognize and ignore POD.

POD content is divided into paragraphs by empty lines. Paragraphs that begin with whitespace characters – tabs or spaces – are considered to be "verbatim paragraphs", and are left completely unformatted; these are used for sample code, ASCII art, etc. Paragraphs that begin with an equal sign are "command paragraphs"; the sequence of alphanumeric characters immediately following the equal sign is treated as a POD directive, and the rest of the paragraph is formatted according to that directive. Some directives also affect the following paragraphs. If a paragraph starts with something besides an equal sign or whitespace, it's considered an "ordinary paragraph".

Both ordinary paragraphs and the contents of command paragraphs are parsed for formatting codes. Formatting in POD is very plain; it's mainly limited to bold, italic, underlined, monospaced, and a few other formats. There is also a code for linking between POD documents or to another section within the same document. Formatting codes consist of either:

- A single uppercase letter, followed by a less-than sign (`<`), the content to be formatted, and a greater-than sign (`>`), e.g. `B<bolded text>`, or
- A single uppercase letter, two or more less-than signs (`<<`), a space, the content to be formatted, another space, and the same number of greater-than signs as were used before, e.g. `B<< bolded text >>`. This form is often used for code snippets containing a greater-than sign, which would

otherwise end the formatting code.

Commands in POD include four levels of headings, bulleted and numbered lists, and commands to mark sections as being in another language. The latter feature allows for special formatting to be given to parsers that support it.

An example of POD in practice is given in the next section.

29.1.4 POD::Usage

Once you're processing your options with something like `Getopt::Long` and you've created your script documentation in POD wouldn't it be handy to use that to automatically create the usage blurb for our script? Let's see...

```
% cat empty
#!/usr/local/bin/perl -w

=head1 NAME

empty - fill in..

=head1 SYNOPSIS

    empty [ -h | -help | -pod ]

=head1 DESCRIPTION

empty is a template.

=head1 OPTIONS

=over 4

=item C<-h> or C<-help>

Displays a short usage message
```

```
=item C<-pod>
```

Display the whole man page

```
=back
```

```
=head1 LIMITATIONS
```

In a sense empty is completely and utterly useless.
But in another sense empty is a real convenience.
who knew?

```
=cut
```

```
use strict;
```

```
#use lib "/home/chicks/src/local_perl_lib";
#use Data::Dumper;
#use DBI;
use Getopt::Long;
use Pod::Usage;
```

```
# constants
```

```
#my $dsn = "DBI:mysql:";
```

```
# get options
```

```
my $usage = 0;
```

```
my $pod = 0;
```

```
GetOptions (
    "h|help"      => \$usage,
    "pod"         => \$pod,
);
```

```
pod2usage() if $usage;
```

```
pod2usage(-verbose => 2) if $pod;
```

```
die "unimplemented";
```

```
% ./empty
```

```
unimplemented at ./empty line 58.
```

```
% ./empty -h
```

```
Usage:
```

```
empty [ -h | -help | -pod ]

% ./empty -?
zsh: no matches found: -?
% ./empty '-?'
Unknown option: ?
unimplemented at ./empty line 58.
% ./empty -pod
EMPTY(1)          User Contributed Perl Documentation          EMPTY(1)

NAME
    empty - fill in..

SYNOPSIS
    empty [ -h | -help | -pod ]

DESCRIPTION
    empty is a template.

OPTIONS
    "-h" or "-help"
        Displays a short usage message

    "-pod"
        Display the whole man page

LIMITATIONS
    In a sense empty is completely and utterly useless. But
    in another sense empty is a real convenience. who knew?

perl v5.8.6          2009-05-18          EMPTY(1)
```

That seems pretty handy for almost no effort. `pod2usage()` also allows you to define the exit value of the script by passing an `-exitval` option. More information can naturally be found using `perldoc POD::Usage` or <http://perldoc.perl.org/Pod/Usage.html>.

29.1.5 AppConfig

AppConfig provides a configuration framework that allows the convenient combination of configuration files, built-in defaults, and command line arguments. The config file format is similar to win32 .ini files. So variables are set by *varname=value* lines split into sections by [*section*] lines. Handy extensions like ~ expansion, variable substitution, # comments, and multi-line values are on all the time or can be enabled.

By default AppConfig requires us to defined the variables we want to use. Once you have an AppConfig handle, you can call the `define()` method with a variable name and hashref of metadata. The must used keys in the metadata hash are `DEFAULT` which defined the default, `ARGCOUNT` which determines whether the variable is treated like an array or a scalar.

To put it together we need to

1. use the module
2. create an AppConfig handle
3. define our variables
4. read configuration file(s)
5. read command line arguments
6. *profit!* (oh wait...)

Let's watch it work:

```
$ cat appconfig
#!/usr/bin/perl -w

use strict;
use AppConfig qw(:argcount);

# create a new AppConfig object
my $config = AppConfig->new();

# define a simple variable
```

```
$config->define(
    foo => {
        ALIAS => 'bar',
        ARGS => '=s', # Getopt::Long hint
        ARGCOUNT => ARGCOUNT_ONE,
    }
);

# same thing
$config->define("foo|bar=s");

$config->define(
    baz => {
        DEFAULT => 'FOO',
        ARGCOUNT => ARGCOUNT_LIST,
    }
);

my $varname = 'foo';
my $value = 'new bar';

# set/get the value
#my $value_copy = $config->get($varname);
my $value_copy = $config->get('foo');
if (defined $value_copy) {
    print "foo starts with $value_copy\n";
} else {
    print "foo starts with no default\n";
}

$config->set( $varname, $value );
$value_copy = $config->get($varname);
print "foo is now set to $value_copy\n";

# shortcut form
$value = "newer bar";
$config->foo($value);
$value_copy = $config->foo;
print "foo is now set to $value_copy\n";
```

```
# read configuration file
$config->file('./appconfig.ini');
print "foo is now set to " . $config->foo . "\n";

# parse command line options
$config->args();      # default to \@ARGV

# advanced command line options with Getopt::Long
$config->getopt();    # default to \@ARGV
print "foo is now set to " . $config->foo . "\n";

# parse CGI parameters (GET method)
$config->cgi($query);      # default to $ENV{ QUERY_STRING }
$ ./appconfig
foo starts with no default
foo is now set to new bar
foo is now set to newer bar
foo is now set to file foo
foo is now set to file foo
```

Not only does AppConfig provide a one stop shop for configuration, it is a great example of one Perl module building on another Perl module.

29.2 File I/O

29.2.1 IO::File

`IO::File` provides an object-oriented interface to file I/O. Each file handle is an object accessed through a reference. If you want to pass file handles around wherever scalars would go this can be convenient. It also provides a seekable interface where possible. Through a call like `IO::File->new_tmpfile()` you can get a file handle to a temporary file also.

29.2.2 IO::Select

Dealing with multiple files simultaneously leads to a variety of interesting difficulties. The classic example is, what if you want to read from 2 different files A and B and you read from A which is on a very slow drive it will block and you won't be able to do any reading from B until it returns. So we need a way to know which filehandles are ready without actually blocking on them which leads to the C `select()` call. The same functionality is available in Perl via `IO::Select`.

```
use IO::Select;

$s = IO::Select->new();

$s->add(\*STDIN);
$s->add($some_handle);

@ready = $s->can_read($timeout);
@ready = IO::Select->new(@handles)->can_read(0);
```

So `add()` accepts filehandles as scalars or typeglobs. The resulting object has a `can_read()` method. Naturally there is a `can_write()` method as well. The timeouts in either case are expressed in seconds, possibly fractional, and the calls will block if it is omitted.

29.2.3 File::Slurp

After a few years of coding Perl you may have written hundreds of file read loops. If you don't have a good reason to retrod that ground, use `File::Slurp`. It'll give it to you as an array or scalar without fuss.

```
use File::Slurp;

my $text = read_file( 'filename' ); # all in one scalar
my @lines = read_file( 'filename' ); # array element = line

write_file( 'filename', @lines );

# similar to Perl 6 syntax
use File::Slurp qw( slurp );
my $text = slurp( 'filename' );
```

What could be simpler? Error handling defaults to `croak()` but can be overridden. Writing can be done atomically, appending to existing file contents, or producing an error if the file already exists.

29.3 Networking

29.3.1 Socket

The `Socket` module provides the classical procedural C style interface to “BSD” sockets. Aside from accepting connections or datagrams, making connections and sending datagrams, the `Socket` module provides utilities such as `inet_aton()`, access to the `/etc/service` file. Generally new code should rely on `IO::Socket` or something built on it.

29.3.2 IO::Socket

`IO::Socket` provides an object interface to creating and using sockets. It is built upon the `IO::Handle` interface and inherits all the methods defined by `IO::Handle`. `IO::Socket` only defines methods for those operations which are common to all types of socket. Operations which are specified to a socket in a particular domain have methods defined in sub classes of `IO::Socket`. `IO::Socket` will export all functions (and constants) defined by `Socket`.

To handle UNIX domain and TCP/IP sockets, we have `IO::Socket::UNIX` and `IO::Socket::INET`, respectively. Some examples:

```
# 3 arguments, quite explicit
$sock = IO::Socket::INET->new(PeerAddr => 'www.perl.org',
                              PeerPort => 'http(80)',
                              Proto    => 'tcp');

# short and sweet
$sock = IO::Socket::INET->new(PeerAddr => 'localhost:smtp(25)');

# accept incoming tcp connections
$sock = IO::Socket::INET->new(Listen    => 5,
                              LocalAddr => 'localhost',
                              LocalPort => 9000,
                              Proto     => 'tcp');
```

```
# connect to local SMTP (port 25)
# one argument is taken as if it were a PeerAddr
$sock = IO::Socket::INET->new('127.0.0.1:25');

# create a udp socket
$sock = IO::Socket::INET->new(PeerPort => 9999,
                              PeerAddr => inet_ntoa(INADDR_BROADCAST),
                              Proto      => udp,
                              LocalAddr => 'localhost',
                              Broadcast => 1
) or die "Can't bind : $@\n";
```

?? more?

29.3.3 Net::Netmask

If you want to do processing with net blocks the Net::Netmask utility can save lots of trouble. For instance:

```
use Net::Netmask;

$block = new Net::Netmask ('192.168.0.1/24')
$block = new Net::Netmask ('192.168.0.1', '255.255.255.0')

# new2 returns error for invalid netmasks
$block = new2 Net::Netmask ('192.168.0.1/24')
$block = new2 Net::Netmask ('192.168.0.1', '255.255.255.0')

print $block;                # a.b.c.d/bits
print $block->base()
print $block->mask()
print $block->hostmask()
print $block->bits()
print $block->size()
print $block->maxblock()
print $block->broadcast()
print $block->next()
print $block->match($ip);
```

```
print $block->nth(1, [$bitstep]);

print 'identical' if $block->sameblock('192.168.0.128/25'); # 0

$newblock = $block->nextblock([count]);
  for $ip ($block->enumerate([$bitstep])) { }
```

More information is naturally available from perldoc `Net::Netmask`.

Fill in more

29.3.4 Net::Ping

Naturally the `Net::Ping` module provides a Perl native interface to ping. An example should make the usage clear:

```
use Net::Ping;

# simple ping
$p = Net::Ping->new();
print "$host is alive.\n" if $p->ping($host);
$p->close();

# specify protocol and hit a lot of hosts
$p = Net::Ping->new("icmp");
foreach $host (@host_array) {
    print "$host is ";
    # second argument is timeout in seconds, e.g., 2s
    print "NOT " unless $p->ping($host, 2);
    print "reachable.\n";
    sleep(1);
}
$p->close();

$p = Net::Ping->new("tcp", 2);
# Try connecting to the www port instead of the echo port
```

```

    $p->{port_num} = getservbyname("http", "tcp");
    while ($stop_time > time()) {
        print "$host not reachable ", scalar(localtime()), "\n"
            unless $p->ping($host);
        sleep(300);
    }
    undef($p);

# Like tcp protocol, but with many hosts
$p = Net::Ping->new("syn");
$p->{port_num} = getservbyname("http", "tcp");
foreach $host (@host_array) {
    $p->ping($host);
}
while (($host,$rtt,$ip) = $p->ack) {
    print "HOST: $host [$ip] ACKED in $rtt seconds.\n";
}

# High precision syntax (requires Time::HiRes)
$p = Net::Ping->new();
$p->hires();
($ret, $duration, $ip) = $p->ping($host, 5.5);
printf("$host [ip: $ip] is alive (packet return time: %.2f ms)\n",
    1000 * $duration
) if $ret;
$p->close();

Go forth and ping.

```

29.3.5 Sys::Hostname

`Sys::Hostname` is pretty self-explanatory: “it tries every conceivable way to get the hostname”. So for example:

```

use Sys::Hostname;
my $host = hostname;

```

would get the hostname into `$host`. Isn't that nice and simple?

Gory details: It attempts several methods of getting the system hostname and then caches the result. It tries the first available of the C library's `gethost-name()`, `'$Config{aphostname}'`, `uname(2)`, `"syscall(SYS_gethost-name)"`, `'hostname'`, `'uname -n'`, and the file `/com/host`. If all that fails it "croak"s. All NULs, returns, and newlines are removed from the result.

29.4 Exercises

29.4.1 Options Processing

1. Use `Getopt::Std`'s `getopt()` call to build a script that takes `-z`, `-x`, and `-q` as arguments. Print out the values that each option receives.
2. Use `Getopt::Std`'s `getopts()` call to build a script that takes `-y`, `-r`, and `-s` as flags. Print out the value of all three flags.
3. Use `Getopt::Long` to create a script. The script should have flag arguments of `-v`, `-verbose`, and `-quiet`. It should also have integer arguments of `-c` and `-i` and string arguments of `-f` and `-o`. Print a usage statement if `-h` or `-help` is passed. Also print an error message plus usage statement if the arguments to `-f` and `-o` are not valid files.
4. Document the previous script using POD.
5. Print out the embedded POD for your usage statement.
6. Convert the previous script to use `AppConfig`. Add a new argument/variable “`lib`” that can receive multiple arguments. Test it to ensure that it works.
7. Add a configuration file to your script.

29.4.2 File I/O

1. Read from two file handles “at the same time”.
2. Read `/etc/passwd` into an array.
3. Turn all of those entries into all caps and write that array into a temporary file.

29.4.3 Networking

1. Open a TCP connection to port 80 and retrieve a web page.

-
2. Create a TCP server and print the current time over each new connection and then close the connection.
 3. Is 192.168.0.99 in 192.168.0.0/24?
 4. Is 172.2.9.45 in 172.2.8.0/20?
 5. What is the netmask for a /12?
 6. What is the broadcast address for 10.9.8.7/6?
 7. Ping `www.yahoo.com`, `mail.yahoo.com`, and `search.yahoo.com`.
 8. What is your hostname? Can you ping your hostname?

29.5 Exercise Answers

Should be written eventually – *contributions welcome*

Chapter 30: Packages and Creating Modules

In this chapter...

You will learn how packages create unique namespaces in Perl and how to take advantage of packages to create modules.

30.1 Create content

This should **get written real soon now** or something like that.

Blah blah blah

30.2 Creating modules

Asdfjkl

30.3 Object Oriented Modules

Chapter 31: Debugging Perl

In this chapter...

We will hopefully help you have an easier time debugging your Perl.

31.1 Create content

This is **not written yet**, sorry.

31.2 Carp module

Nothing yet

31.3 Perl Debugger

Nothing yet

Chapter 32: Win32

In this chapter...

We will show how to use various Win32 modules.

32.1 Win32::EventLog

32.1.1 Win32::EventLog Examples

The following example illustrates the way in which the Win32::EventLog module can be used. It opens the System Event Log and reads through it from oldest to newest. For each record from the source event log it extracts the full text of the entry and prints out the event log message text.

```
use Win32::EventLog;

my $handle = Win32::EventLog->new("System", $ENV{ComputerName})
    or die "Can't open System EventLog";
$handle->GetNumber($recs) or die "can't get number of recs";
$handle->GetOldest($base) or die "can't get index of oldest rec";

while ($x < $recs) {
    $handle->Read(EVENTLOG_FORWARDS_READ|EVENTLOG_SEEK_READ,
        $base + $x, $hashRef
    ) or die "Can't read EventLog entry #$x";
    if ($hashRef->{Source} eq "EventLog") {
        Win32::EventLog::GetMessageText($hashRef);
        print "Entry $x: $hashRef->{Message}\n";
    }
    $x++;
}
```

To backup and clear the event logs on a remote machine do the following:

```
use Win32::EventLog;

my $my_server = '\\my-server'; # your server name here

my ($date) = join('-',
    (
```

```

        ( split /\s+/, scalar localtime )[0,1,2,4]
    )
);

my $dest;

for my $event_log (qw( Application System Security )) {
    $handle = win32::EventLog->new($event_log, $my_server)
        or die "Can't open $event_log event log on $my_server";
    $dest = 'C:\BackupEventLogs\' . $event_log . $date . '.evt';
    $handle->Backup($dest) or warn "Could not backup and clear" .
        " the $event_log event log on \\\\$my_server ($^E)\n";
    $handle->Close;
}

```

32.1.2 Win32::EventLog Reference

This module implements most of the functionality available from the Win32 API for accessing and manipulating Win32 Event Logs. The access to the EventLog routines is divided into those that relate to an EventLog object and its associated methods and those that relate other EventLog tasks (like adding an EventLog record).

32.1.2.1 The EventLog Object and its Methods

The following methods are available to open, read, close and backup EventLogs.

```
win32::EventLog->new(SOURCENAME [,SERVERNAME]);
```

The new() method creates a new EventLog object and returns a handle to it. This handle is then used to call the methods below.

The method is overloaded in that if the supplied SOURCENAME argument contains one or more literal '\' characters (an illegal character in a SOURCENAME), it assumes that you are trying to open a backup eventlog and uses SOURCENAME as the backup eventlog to open. Note that when

opening a backup eventlog, the SERVERNAME argument is ignored (as it is in the underlying Win32 API). For EventLogs on remote machines, the SOURCENAME parameter must therefore be specified as a UNC path.

```
$handle->Backup(FILENAME);
```

The Backup() method backs up the EventLog represented by \$handle. It takes a single argument, FILENAME. When \$handle represents an EventLog on a remote machine, FILENAME is filename on the remote machine and cannot be a UNC path (i.e you must use *C:\TEMP\App.EVT*). The method will fail if the log file already exists.

```
$handle->Read(FLAGS, OFFSET, HASHREF);
```

The Read() method read an EventLog entry from the EventLog represented by \$handle.

```
$handle->Close();
```

The Close() method closes the EventLog represented by \$handle. After Close() has been called, any further attempt to use the EventLog represented by \$handle will fail.

```
$handle->GetOldest(SCALARREF);
```

The GetOldest() method number of the the oldest EventLog record in the EventLog represented by \$handle. This is required to correctly compute the OFFSET required by the Read() method.

```
$handle->GetNumber(SCALARREF);
```

The GetNumber() method returns the number of EventLog records in the EventLog represented by \$handle. The number of the most recent record in the EventLog is therefore computed by

```
$handle->GetOldest($oldest);
```

```
$handle->GetNumber($lastRec);  
$lastRecOffset=$oldest+$lastRec;
```

```
$handle->Clear(FILENAME);
```

The `Clear()` method clears the EventLog represented by `$handle`. If you provide a non-null `FILENAME`, the EventLog will be backed up into `FILENAME` before the EventLog is cleared. The method will fail if `FILENAME` is specified and the file referred to exists. Note also that `FILENAME` specifies a file local to the machine on which the EventLog resides and cannot be specified as a UNC name.

```
$handle->Report(HASHREF);
```

The `Report()` method generates an EventLog entry. The `HASHREF` should contain the following keys:

Computer

The `Computer` field specifies which computer you want the EventLog entry recorded. If this key doesn't exist, the server name used to create the `$handle` is used.

Source

The `Source` field specifies the source that generated the EventLog entry. If this key doesn't exist, the source name used to create the `$handle` is used.

EventType

The `EventType` field should be one of the constants

`EVENTLOG_ERROR_TYPE` = An Error event is being logged.

`EVENTLOG_WARNING_TYPE` = A Warning event is being logged.

`EVENTLOG_INFORMATION_TYPE` = An Information event is being logged.

`EVENTLOG_AUDIT_SUCCESS` = A Success Audit event is being logged (typically in the Security EventLog).

`EVENTLOG_AUDIT_FAILURE` = A Failure Audit event is being logged (typically in the Security EventLog).

These constants are exported into the main namespace by default.

`Category` = The `Category` field can have any value you want. It is specific to the particular `Source`.

`EventID` = The `EventID` field should contain the ID of the message that this event pertains too. This assumes that you have an associated message file (indirectly referenced by the field `Source`).

`Data` = The `Data` field contains raw data associated with this event.

`Strings` = The `Strings` field contains the single string that itself contains NUL terminated sub-strings. These are used with the `EventID` to generate the message as seen from (for example) the Event Viewer application.

32.1.2.2 Other Win32::EventLog functions

The following functions are part of the `Win32::EventLog` package but are not callable from an `EventLog` object.

`GetMessageText(HASHREF);`

The `GetMessageText()` function assumes that `HASHREF` was obtained by a call to `$handle->Read()`. It returns the formatted string that represents the fully resolved text of the `EventLog` message (such as would be seen in the Windows NT Event Viewer). For convenience, the key 'Message' in the supplied `HASHREF` is also set to the return value of this function.

If you set the variable `$Win32::EventLog::GetMessageText` to 1 then each call to `$handle->Read()` will call this function automatically.

32.2 Win32::NetAdmin

You will learn how to manage Windows network groups and users in Perl.

The `win32::NetAdmin` module offers control over the administration of Windows groups and user over a Windows network.

32.2.1 Example

```
# Simple script using win32::NetAdmin to set the login script for
# all members of the NT group "Domain Users". Only works if you
# run it on the PDC. (From Robert Spier <rspier@seas.upenn.edu>)
#
# FILTER_TEMP_DUPLICATE_ACCOUNTS
#   Enumerates local user account data on a domain controller.
#
# FILTER_NORMAL_ACCOUNT
#   Enumerates global user account data on a computer.
#
# FILTER_INTERDOMAIN_TRUST_ACCOUNT
#   Enumerates domain trust account data on a domain controller.
#
# FILTER_WORKSTATION_TRUST_ACCOUNT
#   Enumerates workstation or member server account data on a domain
#   controller.
#
# FILTER_SERVER_TRUST_ACCOUNT
#   Enumerates domain controller account data on domain controller.

use win32::NetAdmin qw(GetUsers GroupIsMember
                       UserGetAttributes UserSetAttributes);

my %hash;
GetUsers("", FILTER_NORMAL_ACCOUNT , \%hash)
    or die "GetUsers() failed: $^E";

foreach ( keys %hash ) {
    my ($password, $passwordAge, $privilege,
```

```

    $homeDir, $comment, $flags, $scriptPath);

    if ( GroupIsMember("", "Domain Users", $_) ) {
        print "Updating $_ ($hash{$_})\n";
        UserGetAttributes("", $_, $password, $passwordAge,
                           $privilege, $homeDir, $comment,
                           $flags, $scriptPath)
            or die "UserGetAttributes() failed: $^E";
        $scriptPath = "dnx_login.bat"; # the new login script
        UserSetAttributes("", $_, $password, $passwordAge,
                           $privilege, $homeDir, $comment, $flags, $scriptPath)
            or die "UserSetAttributes() failed: $^E";
    }
}

```

32.2.2 Win32::NetAdmin provided functions

Note: All of the functions return false if they fail, unless otherwise noted. When a function fails call Win32::NetAdmin::GetError() rather than GetLastError() or \$^E to retrieve the error code.

`server` is optional for all the calls below. If not given the local machine is assumed.

`GetError()`

Returns the error code of the last call to this module.

`GetDomainController(server, domain, returnedName)`

Returns the name of the domain controller for `server`.

`GetAnyDomainController(server, domain, returnedName)`

Returns the name of any domain controller for a domain that is directly trusted by the server.

`UserCreate(server, userName, password, passwordAge, privilege, homeDir, comment, flags, scriptPath)`

Creates a user on server with password, passwordAge, privilege, homeDir, comment, flags, and scriptPath.

`UserDelete(server, user)`

Deletes a user from server.

`UserGetAttributes(server, userName, password, passwordAge, privilege, homeDir, comment, flags, scriptPath)`

Gets password, passwordAge, privilege, homeDir, comment, flags, and scriptPath for user.

`UserSetAttributes(server, userName, password, passwordAge, privilege, homeDir, comment, flags, scriptPath)`

Sets password, passwordAge, privilege, homeDir, comment, flags, and scriptPath for user.

`UserChangePassword(domainname, username, oldpassword, newpassword)`

Changes a users password. Can be run under any account.

`UsersExist(server, userName)`

Checks if a user exists.

`GetUsers(server, filter, userRef)`

Fills userRef with user names if it is an array reference and with the user names and the full names if it is a hash reference.

`GroupCreate(server, group, comment)`

Creates a group.

`GroupDelete(server, group)`

Deletes a group.

`GroupGetAttributes(server, groupName, comment)`

Gets the comment.

`GroupSetAttributes(server, groupName, comment)`

Sets the comment.

`GroupAddUsers(server, groupName, users)`

Adds a user to a group.

`GroupDeleteUsers(server, groupName, users)`

Deletes a users from a group.

`GroupIsMember(server, groupName, user)`

Returns TRUE if user is a member of groupName.

`GroupGetMembers(server, groupName, userArrayRef)`

Fills userArrayRef with the members of groupName.

`LocalGroupCreate(server, group, comment)`

Creates a local group.

`LocalGroupDelete(server, group)`

Deletes a local group.

`LocalGroupGetAttributes(server, groupName, comment)`

Gets the comment.

`LocalGroupSetAttributes(server, groupName, comment)`

Sets the comment.

`LocalGroupIsMember(server, groupName, user)`

Returns TRUE if user is a member of groupName.

`LocalGroupGetMembers(server, groupName, userArrayRef)`

Fills userArrayRef with the members of groupName.

`LocalGroupGetMembersWithDomain(server, groupName, userRef)`

This function is similar `LocalGroupGetMembers` but accepts an array or a hash reference. Unlike `LocalGroupGetMembers` it returns each user name as `DOMAIN\USERNAME`. If a hash reference is given, the function returns to each user or group name the type (group, user, alias etc.). The possible types are as follows:

```
$SidTypeUser = 1;
$SidTypeGroup = 2;
$SidTypeDomain = 3;
$SidTypeAlias = 4;
$SidTypeWellKnownGroup = 5;
$SidTypeDeletedAccount = 6;
$SidTypeInvalid = 7;
$SidTypeUnknown = 8;
```

`LocalGroupAddUsers(server, groupName, users)`

Adds a user to a group.

`LocalGroupDeleteUsers(server, groupName, users)`

Deletes a users from a group.

`GetServers(server, domain, flags, serverRef)`

Gets an array of server names or an hash with the server names and the comments as seen in the Network Neighborhood or the server manager. For flags, see `SV_TYPE_*` constants.

`GetTransports(server, transportRef)`

Enumerates the network transports of a computer. If `transportRef` is an array reference, it is filled with the transport names. If `transportRef` is a hash reference then a hash of hashes is filled with the data for the transports.

`LoggedOnUsers(server, userRef)`

Gets an array or hash with the users logged on at the specified computer. If `userRef` is a hash reference, the value is a semikolon separated string of username, logon domain and logon server.

`GetAliasFromRID(server, RID, returnedName)`

`GetUserGroupFromRID(server, RID, returnedName)`

Retrieves the name of an alias (i.e local group) or a user group for a RID from the specified server. These functions can be used for example to get the account name for the administrator account if it is renamed or localized.

Possible values for RID:

```
DOMAIN_ALIAS_RID_ACCOUNT_OPS
DOMAIN_ALIAS_RID_ADMINS
DOMAIN_ALIAS_RID_BACKUP_OPS
DOMAIN_ALIAS_RID_GUESTS
DOMAIN_ALIAS_RID_POWER_USERS
DOMAIN_ALIAS_RID_PRINT_OPS
DOMAIN_ALIAS_RID_REPLICATOR
DOMAIN_ALIAS_RID_SYSTEM_OPS
DOMAIN_ALIAS_RID_USERS
DOMAIN_GROUP_RID_ADMINS
DOMAIN_GROUP_RID_GUESTS
DOMAIN_GROUP_RID_USERS
DOMAIN_USER_RID_ADMIN
DOMAIN_USER_RID_GUEST
```

`GetServerDisks(server, arrayRef)`

Returns an array with the disk drives of the specified server. The array contains two-character strings (drive letter followed by a colon).

32.3 Win32::NetResource

This module offers control over the network resources of Win32. Disks, printers etc can be shared over a network.

32.3.1 Examples

Enumerating all resources on the network

```
#
# This example displays all the share points in the entire
# visible part of the network.
#

use strict;
use Win32::NetResource qw(:DEFAULT GetSharedResources GetError);
my $resources = [];
unless(GetSharedResources($resources, RESOURCETYPE_ANY)) {
    my $err;
    GetError($err);
    warn Win32::FormatMessage($err);
}

foreach my $href (@$resources) {
    next if ($$href{DisplayType} != RESOURCEDISPLAYTYPE_SHARE);
    print "-----\n";
    foreach (keys %$href){
        print "$_: $href->{$_}\n";
    }
}
```

Enumerating all resources on a particular host

```
#
# This example displays all the share points exported by the
# local host.
#

use strict;
```

```

use Win32::NetResource qw(:DEFAULT GetSharedResources GetError);
if ( GetSharedResources( my $resources, RESOURCETYPE_ANY,
                        { RemoteName => "\\\\" .
                          Win32::NodeName() }
)) {
    foreach my $href (@$resources) {
        print "-----\n";
        foreach(keys %$href) { print "$_: $href->{$_}\n"; }
    }
}

```

32.3.2 Data Types

There are two main data types required to control network resources. In Perl these are represented by hash types.

32.3.2.1 %NETRESOURCE

Key	Value
Scope	Scope of an Enumeration: RESOURCE_CONNECTED, RESOURCE_GLOBALNET, RESOURCE_REMEMBERED.
Type	The type of resource to Enum: <div> <div>RESOURCETYPE_ANY</div> <div>All resources</div> </div> <div> <div>RESOURCETYPE_DISK</div> <div>Disk resources</div> </div> <div> <div>RESOURCETYPE_PRINT</div> <div>Print resources</div> </div>
DisplayType	The way the resource should be displayed. <div> <div>RESOURCEDISPLAYTYPE_DOMAIN</div> <div>The object should be displayed as a domain.</div> </div> <div> <div>RESOURCEDISPLAYTYPE_GENERIC</div> <div>The method used to display the object does not matter.</div> </div> <div> <div>RESOURCEDISPLAYTYPE_SERVER</div> <div>The object should be displayed as a server.</div> </div> <div> <div>RESOURCEDISPLAYTYPE_SHARE</div> </div>

Key	Value
	The object should be displayed as a sharepoint.
Usage	Specifies the Resources usage: RE- SOURCEUSAGE_CONNECTABLE, RE- SOURCEUSAGE_CONTAINER.
LocalName	Name of the local device the resource is connected to.
RemoteName	The network name of the resource.
Comment	A string comment.
Provider	Name of the provider of the resource

32.3.2.2 %SHARE_INFO

This hash represents the SHARE_INFO_502 struct.

Key	Value
netname	Name of the share.
type	type of share.
remark	A string comment.
permissions	Permissions value
maxusers	the max # of users.
current-users	the current # of users.
path	The path of the share.
passwd	A password if one is req'd

32.3.3 Functions

Note: All of the functions return false if they fail.

`GetSharedResources(\@Resources,dwType,\%NetResource = NULL)`

Creates a list in @Resources of %NETRESOURCE hash references.

The return value indicates whether there was an error in accessing any of

the shared resources. All the shared resources that were encountered (until the point of an error, if any) are pushed into @Resources as references to %NETRESOURCE hashes. See example below. The \%NetResource argument is optional. If it is not supplied, the root (that is, the topmost container) of the network is assumed, and all network resources available from the toplevel container will be enumerated.

`AddConnection(\%NETRESOURCE, $Password, $UserName, $Connection)`

Makes a connection to a network resource specified by %NETRESOURCE

`CancelConnection($Name, $Connection, $Force)`

Cancels a connection to a network resource connected to local device \$Name. \$Connection is either 1 - persistent connection or 0, non-persistent.

`WNetGetLastError($ErrorCode, $Description, $Name)`

Gets the Extended Network Error.

`GetError($ErrorCode)`

Gets the last Error for a Win32::NetResource call.

`GetUNCName($UNCName, $LocalPath);`

Returns the UNC name of the disk share connected to \$LocalPath in \$UNCName. \$LocalPath should be a drive based path. e.g. "C:\\[share\\subdir](#)"

Note: \$servername is optional for all the calls below. (if not given the local machine is assumed.)

`NetShareAdd(\%SHARE, $parm_err, $servername = NULL)`

Add a share for sharing.

`NetShareCheck($device, $type, $servername = NULL)`

Check if a directory or a device is available for connection from the net-

work through a share. This includes all directories that are reachable through a shared directory or device, meaning that if C:\foo is shared, C:\foo\bar is also available for sharing. This means that this function is pretty useless, given that by default every disk volume has an administrative share such as "C\$" associated with its root directory.

\$device must be a drive name, directory, or a device. For example, "C:", "C:\dir", "LPT1", "D\$", "IPC\$" are all valid as the \$device argument. \$type is an output argument that will be set to one of the following constants that describe the type of share:

STYPE_DISKTREE	Disk drive
STYPE_PRINTQ	Print queue
STYPE_DEVICE	Communication device
STYPE_IPC	Interprocess communication (IPC)
STYPE_SPECIAL	Special share reserved for interprocess communication (IPC\$) or remote administration of the server (ADMIN\$). Can also refer to administrative shares such as C\$, D\$, etc.

```
NetShareDel( $netname, $servername = NULL )
```

Remove a share from a machines list of shares.

```
NetShareGetInfo( $netname, \%SHARE, $servername=NULL )
```

Get the %SHARE_INFO information about the share \$netname on the server \$servername.

```
NetShareSetInfo( $netname, \%SHARE, $parm_err, $servername=NULL)
```

Set the information for share \$netname.

32.4 Win32::Service

32.4.1.1 Examples

The first script gets a hashref that contains information about all of the services on the current host. It then retrieves status information for each of those into another hashref.

```
use Win32::Service;

my (%service, %status);

Win32::Service::GetServices('', \%services);

foreach my $key (sort keys %services) {
    print "Display Name\t: $key, $services{$key}\n";
    Win32::Service::GetStatus('', $services{$key}, \%status);
    foreach my $part (keys %status) {
        print "\t$part : $status{$part}\n";
    }
}
```

The next script checks the status of NetDDE. If it's already running, it dies with an error. Otherwise, it tries to start it.

```
use Win32::Service;
use Win32;

my %status;

Win32::Service::GetStatus('', 'NetDDE', \%status);
die "service is already started\n"
    if ($status{CurrentState} == 4); # running
Win32::Service::StartService(Win32::NodeName(), 'NetDDE')
    or die "can't start service\n";
```

```
print "Service started\n";
```

32.4.1.2 Functions

Note: All of the functions return false if they fail, unless otherwise noted. If `hostName` is an empty string, the local machine is assumed.

`StartService(hostName, serviceName)`

Start the service `serviceName` on machine `hostName`.

`StopService(hostName, serviceName)`

Stop the service `serviceName` on the machine `hostName`.

`GetStatus(hostName, serviceName, status)`

Get the status of a service. The third argument must be a hash reference that will be populated with entries corresponding to the `SERVICE_STATUS` structure of the Win32 API. See the Win32 Platform SDK documentation for details of this structure.

`PauseService(hostName, serviceName)`

`ResumeService(hostName, serviceName)`

`GetServices(hostName, hashref)`

Enumerates both active and inactive Win32 services at the specified host. The `hashref` is populated with the descriptive service names as keys and the short names as the values.

32.5 Win32::Sound

32.5.1 Quick Sample

A sampling of Perl playing sounds and adjusting the volume:

```
use Win32::Sound;

Win32::Sound::Volume('50%');

# set volume for left and right separately
Win32::Sound::Volume('100%', '50%');

($left, $right) = Win32::Sound::Volume();
Win32::Sound::Volume(0); # mute
Win32::Sound::Volume($left, $right); # restore prior values

Win32::Sound::Play("example.wav") # arbitrary
Win32::Sound::Play("SystemQuestion"); # symbolic

Win32::Sound::Stop();
```

Chapter 33: *NIX cheat sheet

33.1 Some UNIX commands

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

Table 33-1. Simple UNIX commands

Action	Command
Change to home directory	<code>cd</code>
Change to <i>directory</i>	<code>cd <i>directory</i></code>
Change to directory above current directory	<code>cd ..</code>
Show current directory	<code>pwd</code>
Directory listing	<code>ls</code>
Wide directory listing, showing hidden files	<code>ls -al</code>
Showing file permissions	<code>ls -al</code>
Making a file executable	<code>chmod +x <i>filename</i></code>
Printing a long file a screenful at a time	<code>more <i>filename</i></code> or <code>less <i>filename</i></code>
Getting help for <i>command</i>	<code>man <i>command</i></code>

ddd

Chapter 34: Editor cheat sheet

In this chapter...

you will find an editor summary which is laid out as follows:

Table 34-1. 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.

34.1 vi

vi is the classic UNIX editor. It is strange but beautiful. It is very powerful in educated hands and is universally available in the UNIX world.

A version of vi known as vim is available that can easily be installed in Windows and many other strange operating systems. Check out <http://www.vim.org/> for more information.

34.1.1 Running

```
% vi filename
```

34.1.2 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.

34.1.3 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.

34.1.4 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).

34.1.5 Help

:help ENTER might work. If not, then see the man page.

34.1.6 vim

SPONSOR

VOTE

Vim development

for features



the editor

BUY

HELP

LEARN

the Vim book

Uganda

Vim

not logged in ([login](#))

[Home](#)
[Search](#)
[About Vim](#)
[Community](#)
[News](#)
[Sponsoring](#)
[Trivia](#)
[Documentation](#)
[Download](#)
[Scripts](#)
[Tips](#)
[My Account](#)
[Site Help](#)



[What is Vim?](#)
 Vim is a highly configurable text editor built to enable efficient text editing. It is an improved version of the vi editor distributed with most UNIX systems. Vim is distributed free as charityware. If you find Vim a useful addition to your life please consider [helping needy children in Uganda](#).
[What is Vim online?](#)
 Vim online is a central place for the Vim community to store useful Vim tips and tools. Vim has a scripting language that allows for plugin like extensions to enable IDE behavior, syntax highlighting, colorization as well as other advanced features. These scripts can be uploaded and maintained using Vim online.
[listed at inwio.com](#)

News

Vim 7.0.192 is the current version

Vim presentation in Mountain View

[2007-02-05] Tuesday, February 13th, I will be giving a presentation at the Google offices in Mountain View. The title is "Seven habits for effective text editing, 2.0". It will start at 7 pm. More information can be found on the [Google code site](#). For instructions how to get there click on "Mountain View headquarters". But pay attention to the building number 41, there are many Google buildings :-). Hope to see many of you there! (*Bram Moolenaar*)

[@vim.org email back](#)

[2007-02-04] The problem in the @vim.org email has been solved, we're back! (*Bram Moolenaar*)

[more news...](#)
[Find Vimmers on Frappri](#)
[DVD and video about Vim's charity project](#)

Recent Script Updates

1,784 scripts, 1,702,071 downloads

[2007-02-16] [surround.vim](#) : Delete/change/add parentheses/quotes/XML-tags/much more with ease (1.23) xmap rather than vmap, to avoid interfering with select mode. surround_insert_tail to specify a universal suffix for use in insert mode. - *Tim Pope*

[2007-02-15] [polycl.vim](#) : Polyhedra CL syntax (0.9.1) Major fixes (typos). Fixed string escapes. Started highlighting of operators and split of reserved word in multiple classes. - *Olivier Mengué*

[2007-02-15] [polycfg.vim](#) : Polyhedra configuration syntax (1.0) Initial upload - *Olivier Mengué*

[2007-02-15] [fcsh tools](#) : you can compile .as and .mxm files from vim via fcsh : Flex Complier SHell (0.2) added more quotes. - *mike rowe*

[more recent](#) | [most downloaded](#) | [top rated](#)

Recent Tip Additions

1,304 tips, 3,657,456 tip views

[2007-02-06] [tip #1504](#) - External commands on Windows (Tim Keating)

[2007-02-02] [tip #1501](#) - substitute last search (Jerome)

[2007-02-01] [tip #1500](#) - By default, when opening files in Mac OS X, a new vim window is opened. This shows you how to have only one window. (edit in a single window in Mac OS X)

[2007-02-01] [tip #1499](#) - Jump back to spell checked words (john AT beever DOT nl)

[more recent](#) | [most viewed](#) | [top rated](#)

If you have questions or remarks about this site, visit the [vimonline development](#) pages. Please use this site responsibly.

Questions about [Vim](#) should go to vim@vim.org after searching [the archive](#). Help Bram [help Uganda](#). [stats](#)

Special thanks to our sponsors:

[Idealo - Preisvergleich in Österreich](#)
[Test und Preisvergleich](#)
[Price Comparison](#)
[Yatego Shopping](#)






Illustration 8: <http://www.vim.org/>

34.2 pico

pico is the editor from pine turned into an external command. pine is no longer supported by some Linux distributions so you may have to type "nano" to get "pico", but you can always make an alias.

34.2.1 Running

```
% pico -w filename
```

34.2.2 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.

34.2.3 Exiting

Follow the menu bar, if you are in the midst of a command. Use **CTRL-X** from the main menu.

34.2.4 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

34.2.5 Help

CTRL-G from the main menu, or just read the menu bar.

34.3 joe

34.3.1 Running

`% joe filename`

34.3.2 Using

- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-K** then **S** to save.

34.3.3 Exiting

- **CTRL-C** to exit without save.
- **CTRL-K** then **X** to save and exit.

34.3.4 Gotchas

Nothing in particular.

34.3.5 Help

CTRL-K then **H**.

34.4 jed

34.4.1 Running

% jed

34.4.2 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.

34.4.3 Exiting

CTRL-X then **CTRL-C** to exit.

34.4.4 Gotchas

Nothing in particular.

34.4.5 Help

- Read the menu bar at the top.
- Press **ESC** then **?** then **H** from the main menu.

Chapter 35: ASCII Pronunciation Guide

In this chapter...

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 35-1. ASCII Pronunciation Guide

Character	Pronunciation
!	bang, exclamation
*	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, squiggles, or squiggly brackets
	pipe, bar, or vertical bar
~	tilde (pron. "til-duh"), wiggle
`	backtick, backquote (below ~)

Chapter 36: HTML Cheat Sheet

In this chapter...

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	<code><P> ... </P></code>
Heading level 1	<code><H1>This is a level 1 heading</H1></code>
Heading level 2	<code><H2>This is a level 2 heading</H2></code>
Heading level 3	<code><H3>This is a level 3 heading</H3></code>
Heading level 4	<code><H4>This is a level 4 heading</H4></code>
Unordered (bulleted) list	<pre> List item 1 List item 2 List item 3 </pre>
Ordered (numbered) list	<pre> List item 1 List item 2 List item 3 </pre>
Table	<pre> <TABLE BORDER> <TR> <-- "table row" -- > <TH>Heading column 1</TH> <TH>Heading column 2</TH> <TH>Heading column 3</TH> </TR> <TR> <-- "table row" -- > <TD>row 1, column 1</TD> <TD>row 1, column 2</TD> <TD>row 1, column 3</TD> </TR> <TR> <-- "table row" -- > <TD>row 2, column 1</TD> <TD>row 2, column 2</TD> <TD>row 2, column 3</TD> </TR> </TABLE> </pre>
Horizontal rule	<code><HR></code>
Anchor tag (hyper-text link)	<code>De- scriptive text</code>

For more information...

- HTMLhelp.org (<http://htmlhelp.org/>)
- The World Wide Web Consortium (W3C) (<http://w3.org/>)

Chapter 37: The Regex Coach

In this chapter...

What follows is the nearly verbatim extract of <http://www.weitz.de/regex-coach> which you can go to directly if you're viewing this online, but for those die-hard fans of killing trees to make reading easier (such as your humble author), here's some information on a neat utility.

37.1 Abstract

The Regex Coach is a graphical application for Windows which can be used to experiment with (Perl-compatible) regular expressions interactively. It has the following features:

- It shows whether a regular expression matches a particular target string.
- It can also show which parts of the target string correspond to captured register groups or to arbitrary parts of the regular expression.
- It can "walk" through the target string one match at a time.
- It can simulate Perl's `split` and `s///` (substitution) operators.
- It tries to describe the regular expression in plain English.
- It can show a graphical representation of the regular expression's parse tree.
- It can single-step through the matching process as performed by the regex engine.
- Everything happens in "real time", i.e. as soon as you make a change somewhere in the application all other parts are instantly updated.



If you find this software useful then please consider making a small donation towards the ongoing development costs. Website hosting costs money, as do compilers and development tools.

37.2 Contents

- [Download and installation](#)
 - [Older versions, Linux, FreeBSD, Mac](#)
- [License](#)
- [Support, bug reports, mailing list](#)
 - [How to report bugs](#)
- **[Quick start tutorial](#)**
(An [Italian version](#) is available thanks to Lorenzo Marcon)
- [How to use *The Regex Coach*](#)
 - [The main panes](#)
 - [The message areas](#)
 - [Highlighting selected parts of the match](#)
 - [The highlight buttons](#)
 - [The highlight messages](#)
 - [Walking through the target string](#)
 - [Narrowing the scan](#)
 - [The info pane](#)
 - [The parse tree](#)
 - [Replacing text](#)
 - [Splitting text](#)
 - [Single-stepping through the matching process](#)
 - [Modifiers](#)
 - [Resizing](#)
 - [Saving to and loading from files](#)
 - [Autoscroll](#)
- [Known bugs and limitations](#)
- [Technical information](#)
- [Compatibility with Perl](#)
- [Acknowledgements](#)

37.3 Download and installation

The Regex Coach together with this documentation can be downloaded from <http://weitz.de/files/regex-coach.exe>. The current version is 0.9.1 - see the [changelog](#) for what's new. The file (an installer) is about 2MB in size.

You *should* use Windows 2000 or Windows XP with all [updates and service packs](#) installed. The program *might* work with older or unpatched Windows versions, but don't expect support for these configurations. See also [below](#).

You also **must** have the Microsoft runtime library `msvcr80.dll` installed. If you don't have it or if you aren't sure, you can get it from <http://www.microsoft.com/downloads/details.aspx?familyid=32BC1BEE-A3F9-4C13-9C99-220B62A191EE&displaylang=en>.

If you have a previous version (0.8.5 or earlier) of *The Regex Coach* installed, *uninstall* it first before you install the new version! If you haven't done this, and the new application won't start, remove the file `The Regex Coach.exe.manifest` from the application directory.

37.3.1 Older versions, Linux, FreeBSD, Mac

Beginning with version 0.9.0, there will no longer be a Linux version of *The Regex Coach* - too few people were using it, and it's simply too much work for me to maintain both versions. You can still download the last (now unsupported) Linux release from <http://weitz.de/files/regex-coach-0.8.5.tgz> - it will also run on FreeBSD, see documentation.

If you have an older version of Windows and the current version of *The Regex Coach* doesn't work for you, you can try the last release which was built with [LispWorks 4.4.6](#) - it is at <http://weitz.de/files/regex-coach-0.8.5.exe>. If that works for you - fine. Don't expect [support](#) or updates, though.

There is no Mac version and I have no plans to release one. Sending me email and begging for it won't change that. And, no, I don't want to open source the application or send the source code to *you* privately - no need to ask...

License

The Regex Coach is Copyright © 2003-2006 Dr. Edmund Weitz - All Rights Reserved.

The Regex Coach is free for private or non-commercial use but if you like and use it it'd be nice if you could [donate a small amount](#) to fund further development. *The Regex Coach* is also free for commercial *use* but you are **not** allowed to re-distribute it and/or charge money for it without written permission by the author - email me at edi@weitz.de for details.

The program is provided 'as is' with *no* warranty - use at your own risk.

37.4 Support, bug reports, mailing list

If you want to be notified about new releases of *The Regex Coach* please subscribe to the "regex-coach" mailing list using the web frontend at <http://common-lisp.net/mailman/listinfo/regex-coach>. You can **search** the mailing list archives using [this Google Custom Search Engine](#).

You should also use this list for questions, bug reports, and feature requests.

37.4.1 How to report bugs

If you've found a bug in *The Regex Coach*, I'm happy if you report it and I'll try to fix it. However, please follow the following procedure:

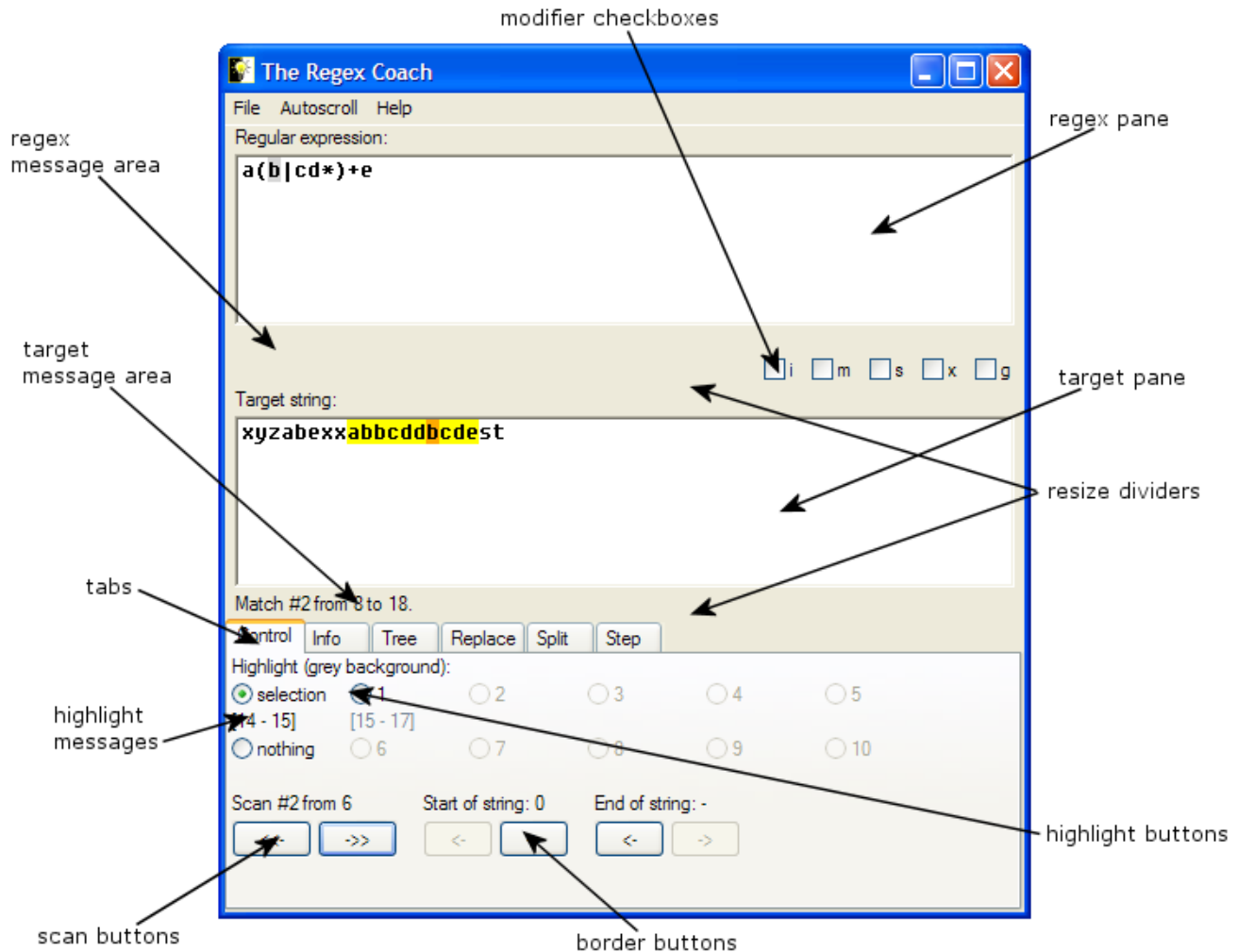
- Make sure you're using the [latest version](#) of *The Regex Coach* on Windows. Older versions and other operating systems are no longer supported.
- Make sure you have `msvcr80.dll` installed - see [above](#).
- Make sure you *don't* have the old manifest file anymore - see [above](#).
- Provide information about the Windows version (including service pack) you're using.
- Try to reduce the problem you're encountering to a simple, self-contained test case, so that I can reproduce the bug easily.
- Send bug reports to the [mailing list](#) and **not** to me privately. I might simply ignore reports not sent to the list.
- If you have five minutes, read [this text](#) by Simon Tatham.

If you think this is asking too much, please keep in mind that you get *The Regex Coach* for free and nobody pays me for fixing bugs or answering questions. If it's too much work for you to send a decent bug report to the right place, then I think it's only fair if I consider it too much work for me to answer.

37.5 How to use *The Regex Coach*

The Regex Coach enables you to try out the behaviour of Perl's regular expression operators (namely `m//`, `s///`, and `split`) interactively and in "real time", i.e. as soon as you make changes somewhere the results are instantly displayed. You can also query the regex engine about selected parts of your regular expression and watch how it parses your input.

Of course, this application should also be useful to programmers using Perl-compatible regex toolkits like [PCRE](#) (which is used by projects like [Python](#), [Apache](#), and [PHP](#)) or [CL-PPCRE](#). Also, [Java's regular expressions](#) and those of [XML Schema](#) are very similar to Perl's.



The following descriptions will use the notions introduced by this annotated screenshot.

37.5.1 The main panes

The main area of the application is inhabited by two panes which are always visible. Both behave like simple editors, i.e. you can type text into them and modify it. You can also copy and paste text between these panes and other applications. On Windows, the keybindings resemble those of typical Windows editors, on Linux the keybindings are those of [GNU Emacs](#). (If you have never used Emacs you might know a couple of these keybindings from the bash shell.) You can use the TAB key to switch between these editors. This will also cycle through the [replacement pane](#) if it's visible.

The upper pane is the [regex pane](#). Here you'll type the regular expression you want to investigate.

The second pane is the [target pane](#). Here you'll type the text (the *target string*) the regular expression will try to match.

If there's a match, the part of the target string that matched will be emphasized by a yellow background. (If you also check the 'g' [modifier checkbox](#) all matches will be emphasized - the ["current"](#) one in yellow, the others in green.)

37.5.2 The message areas

Both of the afore-mentioned panes have *message areas* directly below them. The [regex message area](#) is usually empty but it will show an error message in red letters if the regular expression isn't syntactically correct. It'll also show a warning in grey letters if the content of the regex pane ends with whitespace because this might not be what you want. You can of course ignore this warning if you typed the whitespace characters on purpose.

The [target message area](#) will show the extent of the match (or notify you that there isn't a match at all). This is particularly useful if there's a zero-length match because you won't see any highlighted characters in the target pane in this case. The message "Match from *n* to *m*" means that the characters starting from position *n* up to *m* (exclusively) belong to the match. The first character of the string is character 0 (zero) as usual.

37.5.3 Highlighting selected parts of the match

If there's a match you can highlight selected parts of the match which are shown

in orange. The default setting is to reflect the *selection* you've made in the regex pane. It works like this: If you've selected a valid subexpression of the regular expression in the regex pane the corresponding part of the target string is shown in orange. You see an example in the [screen shot](#) above where the 'b' in the regular expression was selected which corresponds to the fourth 'b' in the target string.

If you've made an invalid selection the [selection highlight button](#) is disabled. You'll also see a message about your selection being invalid in the [info pane](#).

If you have no idea what a "valid subexpression" of the regular expression could be consider the following rule of thumb: Every part of the regular expression which can be wrapped in a non-capturing group - i.e. with `(?: ...)` - without altering the meaning of the expression is valid.

(A more precise description of this would be: Consider the [parse tree](#) of the regular expression and assume that every leaf of the tree which is a string is further divided into the single characters which together constitute the string. Now, every contiguous part of the regular expression which can be completely and exactly covered by nodes of the parse tree is a valid subexpression.)

37.5.4 The highlight buttons

Apart from highlighting the part of the target string which corresponds to the selected area in the [regex pane](#) you can also highlight the parts which correspond to captured register groups (enclosed by parentheses) in the regular expression. This is done by selecting one of the [highlight buttons](#). These are only enabled if there are any captured registers.

Press the "nothing" button to disable highlighting.

37.5.5 The highlight messages

Each of the [highlight buttons](#) has a small [highlight message](#) associated with it (similar to the [message area](#) of the target pane) which shows which part would be highlighted if the corresponding button were selected. Again, this is particularly useful in the case of zero-length (sub-)matches.

37.5.6 Walking through the target string

Usually, the application will try to find the first match beginning from position 0

of the target string. You can use the [scan buttons](#) to move forward (or backward) one match at a time if there's more than one match. (This is how the Perl regex engine would behave in case of 'global' matches - i.e. those with a 'g' [modifier](#) - or if you apply the [split](#) operator.)

The headline above the scan buttons which usually says "Scan from 0" will change accordingly showing a message like "Scan #*n* from *m*" which means that the regex engine is trying to find the *n*th match starting at character *m* of the target string. The [target message area](#) will be changed as well - it'll say "Match #*n* from *k* to *l*" instead of "Match from *k* to *l*" (or it'll say "No further match" instead of "No match" if you've pressed the scan forward button too often).

37.5.7 Narrowing the scan

By using the [border buttons](#) you can narrow the scan to a part of the target string. This effectively hides characters from the start and/or end of the target string from the regex engine. The characters which are masked thusly are covered with a dark grey color in the [target pane](#). Note that the effect of the [scan buttons](#) is reset by the border buttons.

37.5.8 The info pane

Choosing the "Info" [tab](#) will reveal the *info pane* which is an area where the application tries to explain what the regular expression is supposed to do in plain English. If you've [selected](#) a part of the regular expression only this part will be explained.

37.5.9 The parse tree

If you select the "Tree" [tab](#) you'll see a (simplified) graphical representation of the parse tree of the regular expression. This is how the regex engine "sees" the expression and it might help you to understand what's going on (or why the regular expression isn't interpreted as you intended it to be).

37.5.10 Replacing text

By choosing the "Replace" [tab](#) you'll open up an area with two panes. The first one includes a simple editor like the ones in the [main panes](#). Here you can type a replacement string which acts like the second argument to Perl's `s///` (substitution) operator. The second pane will show the result of the substitution. The contents of these panes are meaningless if the regular expression has syntactical er-

rors.

Note that you'll have to use `"\"`, `"\"`, `"\"` and `"\n"` instead of Perl's `"$&"`, `"$`"`, `"$'"` and `"$n"` - see the [CL-PPCRE documentation](#) for the gory details.

37.5.11 Splitting text

The "Split" [tab](#) will reveal a pane which shows the result of applying Perl's `split` operator to the target string. As this result is usually an array of strings the elements of this array are visually divided by vertical lines the size of a space character. (This implies that two vertical lines in a row denote that there's a zero-length string between them. And it also follows that the array has only one element if there's no vertical line at all.)

You can use the radio buttons below the pane to select another divider if the vertical line happens to be a part of your target string. But note that choosing the "block" option might significantly slow down the program if your target strings are long.

You can type a non-negative integer into the "Limit" field. This corresponds to the optional third argument to Perl's `split` operator.

37.5.12 Single-stepping through the matching process

Finally, the "Step" [tab](#) will lead you to two panes which have the same content as the two [main panes](#). However, here you can watch the regex engine "at work". This is best explained with an example, so see the [corresponding part of the tutorial](#).

Note that many of the optimizations done by the [CL-PPCRE](#) engine are turned off here for pedagogical reasons. (For example, when trying to match the regex `a*abc` against the target string `aaaabd` the "real" engine wouldn't even start because it'll first use a Boyer-Moore-Horspool search to check if the constant string `abc` is somewhere in the target.) Some of them remain, however: The engine will only try to match from position 0 if the regex starts with `.*` and is in single-line mode. Also, as you'll see, the stepper tries to match constant strings as a whole (instead of single characters which would be quite boring).

37.5.13 Modifiers

Pressing one of the [modifier checkboxes](#) is equivalent to using the corresponding

modifier character in Perl. For example, the "i" checkbox toggles between case-sensitive and case-insensitive matching. Note that the "g" ('global') modifier only affects the [replacement](#) operation - it has no effect on the match itself. If it's enabled other matches the engine would find are highlighted in green in the [target pane](#), though.

37.5.14 Resizing

You can resize the application window as usual by dragging the lower right corner. But you can also resize the panes relative to each other by dragging one of the [resize dividers](#). These aren't visible in the Windows version but you'll note that the cursor changes if you position the mouse above them. There's also a resize divider between the two [replacement panes](#). *The Regex Coach* will remember the size and position of its main window between two invocations.

37.5.15 Saving to and loading from files

If one of the two [main panes](#) has the focus you can - from the file menu - insert the contents of a text file into this pane or save the contents of this pane to disk. The latter can also be done by pressing Ctrl-s (or Ctrl-x Ctrl-s on Linux). The contents of these two panes will also remain persistent between two invocations of *The Regex Coach*.

Note: Due to the way Motif works, the file menu can't be used like this on Linux. Instead you can use the Emacs key sequences Ctrl-x Ctrl-w and Ctrl-x i.

37.5.16 Autoscroll

The Regex Coach has an *Autoscroll* feature which can be switched on and off via the corresponding menu. If *Autoscroll* is on, then each time the target string is parsed the scrollbar of the target pane will be moved such that the start (or end - depending on what you've chosen) of the match is visible more or less in the middle of the pane. If you've chosen to [highlight](#) specific parts of the match, then the scrollbar will move to the start or end of the highlighted region instead. This is of course only meaningful if the target string is too large to fit into the pane.

No automatic scrolling occurs while the target pane has the input focus.

37.6 Known bugs and limitations

The regex engine might give up with a stack overflow on relatively long regular expressions. (This will happen much earlier as with [CL-PPCRE](#) alone as the parsing process is interwoven with code specific to *The Regex Coach*.) Although maybe counter-intuitive, it might help to add some non-capturing groups, i.e. "aa...abb...b" (with enough characters inbetween) might fail while "(?:aa...a)(?:bb...b)" doesn't.

Also, there seem to be problems with Eastern European versions of Windows, specifically with "character set 1250" or similar. Sorry, I currently don't have the time and resources to investigate this any further.

If you encounter any other bugs or problems please send them to the [mailing list](#).

37.7 Technical information

The Regex Coach is written in [Common Lisp](#) and was developed using the [Lisp-Works](#) development environment. The regex engine used is [CL-PPCRE](#).

It might be worthwhile to note that due to the dynamic nature of Lisp *The Regex Coach* could be written without changing a single line of code in the CL-PPCRE engine itself although the application has to track information and query the engine *while* the regular expressions is parsed and the scanners are built. All this could be done 'after the fact' by using facilities like `defadvice` and `:around` methods. Imagine [writing this application in Perl](#) without touching Perl's regex engine... :)

Also, thanks to LispWork's cross-platform CAPI toolkit the code for the Windows and Linux versions is nearly identical without any platform-specific parts (except for some lines regarding different fonts and keybindings).

37.7.1 Compatibility with Perl

See the [CL-PPCRE documentation](#).

37.8 Acknowledgements

The script to compile the Windows installer was kindly provided by [Ian H.](#) The icon for the Windows application was created by André Derouaux. The PNG included with the Linux distribution was contributed by John Troy Hurteau and is based on André's icon. The Lisp logo was designed by [Manfred Spiller](#). Thanks to Alex Wood for RPM information. Thanks to Jim Prewett for FreeBSD info.

Brigitte Bovy from LispWorks ("Xanalys" at that time) support helped with the tricky interaction between the editor panes. I also got a couple of helpful tips from the Lispworks mailing list, specifically from Jeff Caldwell, John DeSoi, David Fox, and Nick Levine.

Thanks to the guys at "[Café Olé](#)" in Hamburg where I wrote most of the code.

Development of the *The Regex Coach* has been supported by [Euphemismen.de](#).

Chapter 38: GPL2

In this chapter...

You will find our license.

38.1 GNU General Public License

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

38.1.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses,

in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

38.1.2 Terms and Conditions for Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty)

and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a)** Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- b)** Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c)** Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special ex-

ception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other prop-

erty right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST

OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Chapter 39: Acknowledgements

In this section...

I will try to thank a few of the folks and projects that made this possible

39.1 Folks

Thanks to John Lundin for vast contributions of systems administration, content comments, and wonderful stir fries.

Thanks to Stephen Johnson for supporting my instruction and content creation efforts for many years now. If it weren't for Stephen I would never have taught this course for US News & World Report, Circuit City, or a lot of other folks.

Thanks to all of the folks who have survived my instruction of this course and others. Your ideas, comments, complaints, and foolishness have all helped make this class what it is.

Thanks to Mark Whittington for automotive wisdom and other random surprises. Sigh.

Thanks to Kirrily "skud" Robert for creating the DocBook version of this content and sharing it with the world. If only DocBook weren't such a pain. (Writing LISP to make style sheets? Ick.)

Thanks to Cynthia Manuel, Carl Hicks, Thomas St. Jacques, Jason Werner, Matt Solovay and the Collies for many various non-technical contributions.

39.2 Projects

OpenOffice.org for providing a nice free word processor.

dia for easy ERD editing.

Fedora for a damn fine desktop Linux.

CentOS and Red Hat for a damn fine server Linux.

TWiki for a mighty fine wiki. Written in Perl naturally.

Perl for being there to teach. Larry, Randal, and a cast of thousands work together to produce art and technology that looks less like a committee product than most geeks would expect.

